


Article

# Cognification of Program Synthesis—A Systematic Feature-Oriented Analysis and Future Direction

Ahmad F. Subahi 

Department of Computer Science, University College of Al Jamoum, Umm Al Qura University,  
Mecca P.O.Box 715, Saudi Arabia; AFSubahi@uqu.edu.sa; Tel.: +966-5500-5585-6

Received: 6 March 2020; Accepted: 1 April 2020; Published: 12 April 2020



**Abstract:** Program synthesis is defined as a software development step aims at achieving an automatic process of code generation that is satisfactory given high-level specifications. There are various program synthesis applications built on Machine Learning (ML) and Natural Language Processing (NLP) based approaches. Recently, there have been remarkable advancements in the Artificial Intelligent (AI) domain. The rise in advanced ML techniques has been remarkable. Deep Learning (DL), for instance, is considered an example of a currently attractive research field that has led to advances in the areas of ML and NLP. With this advancement, there is a need to gain greater benefits from these approaches to cognify synthesis processes for next-generation model-driven engineering (MDE) framework. In this work, a systematic domain analysis is conducted to explore the extent to the automatic generation of code can be enabled via the next generation of cognified MDE frameworks that support recent DL and NLP techniques. After identifying critical features that might be considered when distinguishing synthesis systems, it will be possible to introduce a conceptual design for the future involving program synthesis/MDE frameworks. By searching different research database sources, 182 articles related to program synthesis approaches and their applications were identified. After defining research questions, structuring the domain analysis, and applying inclusion and exclusion criteria on the classification scheme, 170 out of 182 articles were considered in a three-phase systematic analysis, guided by some research questions. The analysis is introduced as a key contribution. The results are documented using feature diagrams as a comprehensive feature model of program synthesis showing alternative techniques and architectures. The achieved outcomes serve as motivation for introducing a conceptual architectural design of the next generation of cognified MDE frameworks.

**Keywords:** cognifying model-driven engineering; model transformations; program synthesis; software engineering; code generation; domain specific languages; architectural design

## 1. Introduction

Since the early days of computer science, the automatic generation of correct, complete, and executable program code from high-level logical specifications has been a grand ambition. Program synthesis is defined as the automatic process of constructing executable programs that satisfy a given high-level specification. It is also considered a type of code translation that lowers the abstraction level of program code [1], unlike compilers which only accept correctly written high-level source code to be able to extract certain low-level facts after performing direct translation of the syntax and generate a platform-specific machine code to run.

Synthesizers, on the other hand, can accept various types of high-level specifications, such as logic formula, domain-specific language, grammar, natural language, and even partial program source code. Instead of applying direct translation into low-level executable code, a suitable search technique is applied over some space [2] in order to achieve the solution language (final code) or proof (logical

representation). Program synthesis brings many benefits to software engineers and developers when it is applied to the software system development lifecycle. It helps several development tasks, such as model checking, testing and code repairing, to be accomplished without using advanced programming skills. Additionally, it also supports developers in their everyday tasks such discovering or optimizing new algorithms [1].

Recently, the interest in program synthesis has quickly increased. On the other hand, with the rapid evolution of data and learning-based techniques, such as the recent trend toward deep learning techniques, many software development processes and activities at each level of Software Development Lifecycle (SDLC) have become more intelligent. There are two possible integration methods between a search-based program synthesis paradigm and machine learning, including machine learning (ML) and deep learning (DL), that bring benefits from both fields of computer science (software engineering and machine learning). The first method is to use a learning approach to guide traditional synthesized search techniques, which can restrict or reduce the search space to enhance the overall performance of the synthesizer. On the other hand, learning methods might be used directly to search through the program space.

In this paper, a feature model of program synthesis that results from a deep and structured systematic domain analysis is introduced as the first contribution of this investigation. Distinguishable features for the categorization of existing program synthesis approaches and frameworks are presented through the application of systematic domain analysis to published works that are reviewed from various sources such as the Institute of Electrical and Electronic Engineers (IEEE) Xplore, the Association for Computing Machinery (ACM) Digital Library, Science Direct, and Springer. The results of the domain analysis are documented using a technique that has been widely adopted in software engineering—feature-based modelling through several feature diagrams.

The results of this work can help developers and software engineers who are interested in automatic code synthesis and generation to understand program synthesis principles and decide the degree to which it is possible to cognify the program synthesis process as a part of various applications, especially automatic code generation, in the forthcoming generation of model-driven engineering (MDE) frameworks. Additionally, the conducted systematic review as well as the proposed feature model guide us in introducing a conceptual architectural design of what we call the “cognified code generation framework”, as a second contribution. This framework, which is discussed in Section 8, supports sophisticated model transformations and the code generation of multiple kinds of modeling language.

## 2. Brief History

Research on program synthesis has been an active area of research since the early 1960s. Computer scientists within different disciplines, especially the Artificial Intelligence and Software Engineering communities, have been involved in the program synthesis domain of research since its inception. Program synthesis is used in various engineering activities when adopting formal methods in the software development lifecycle, such as code transformations, verification, and testing. The suitable synthesis techniques for these types of engineering tasks are those that belong to the so-called old school strategy in the deductive program synthesis.

It is worth mentioning here that program synthesis began with Alonzo Church [3] who used restricted recursive arithmetic (SIS) to specify the synthesis problem. His problem shaped the first-generation synthesis approaches and is now known as Church’s problem, or Realizability [4]. Richard Büchi and Lawrence Landweber [5], and then Michael O Rabin [6], introduced solutions to the Church problem based on the infinite game theory and tree automata, respectively [4], in the next decade.

Additionally, several influential classical approaches were introduced over time during the 1970s and 1980s, and these have served as the basis for many recent program synthesis approaches and tools, such as the Automatic Programming System (THESYS) approach that was introduced by Phillip D.

Summer in 1977 [7]. It is a knowledge-dependent approach that can deal with structural list problems in programs written in Lisp language only. It was also restricted to the induction of a single, linear recursive equation with a single input list. Furthermore, Amir Pnueli [8] successfully introduced temporal logic to program synthesis and formal verification and specification language formalization. Many influential works based on temporal logic followed, such as [9,10] in the 1980s, [11] in the 1990s, and [12] in 2000, as mentioned in [4].

### 3. Transformational Systems and Program Synthesis

In the software engineering domain, a computer program that accepts some type of input, performs many computations, and produces some type of output in different forms, or levels of abstraction of that input, is called a transformational system. There are several applications in the software development lifecycle where transformational systems can be used, such as compiler construction, software documentation, and code visualization. These applications can be distinguished based upon two main software engineering (SE) scenarios, namely, translation scenarios and rephrasing scenarios. Translation is the process of mapping a source language into a different target one, whereas rephrasing is the process of mapping a source language into a different form of the same language [1].

The classification of transformational systems becomes remarkable in model-based software engineering (MBSE) and Model-driven Engineering (MDE) approaches, where formal semantics play critical roles in the development process, including validation, verification, and automatic code generation. In this domain, transformational systems are distinguished based on different factors [13,14]. One of these factors is the type of metamodel of the source and the target. The formal automated transformational process (algorithm) that is applied to a source artifact (e.g., a system code or model) to produce a target one, where both artifacts are defined and expressed by the same metamodel, is considered to be (endogenous) transformation. On the other hand, when the metamodel of the source artifact differs from the metamodel of the target one, the transformational process is considered to be exogenous transformation or translation. Specifically, applications of transformation have emerged in MDE via a variety of processes such as model modification, inference, weaving, normalization, and model optimization. However, translation applications can clearly be seen in different MDE processes such as model refinement, model migration, automatic code generation, and reverse engineering [1,13,14].

Program synthesis is a prominent example of a translation scenario where the source program (code) defined by a particular metamodel is translated into a lower level of abstraction that is described by another metamodel. Synthesis can be used in two different forms, namely, program refinement and program compilation [1]. For program refinement, the translation process is applied at a high-level specification to obtain a detailed, low-level implementation code. On the other hand, program compilation is considered to be the series of translation processes that translate a high-level executable code into a platform-specific machine one [1].

### 4. Methodology

The commonly known Systematic Literature Review approach was applied. A structured domain analysis process, guided by several research questions, was defined to review and evaluate research relevant to the domain of program synthesis. The defined process included (1) research questions (RQs), (2) data sources, (3) the search strategy, (4) inclusion searching terms (IC), (5) exclusion criteria (EC), and (6) the categorization scheme (CS).

The feature-based modelling technique was adopted to identify all characteristics of program synthesis systems during the review and analysis of the associated literature. The technique was also used to illustrate and document all findings by means of feature diagrams (FDs). The findings of this work are all possible alternative designs and techniques that could be used to construct MDE code generation tools that aid in the development of advanced program synthesis techniques.

#### 4.1. Strategy of Domain Analysis

In order to cover all possible alternative designs and techniques in the core dimensions of program synthesis (user intent, program space and search techniques and synthesis applications), the domain analysis is organized in three levels of reviewing and evaluating research papers. The analysis phases are listed as follows:

1. The first level of analysis identifies the differences between paradigms of program synthesis in terms of the synthesizing approach adopted in each paradigm. This level does not compare the detailed features of the program synthesis approach and its applications.
2. The second level compares different program synthesis approaches in terms of the main three dimensions, namely, user intent, search space and search techniques.
3. The third level identifies the applications of program synthesis from a software engineering perspective.

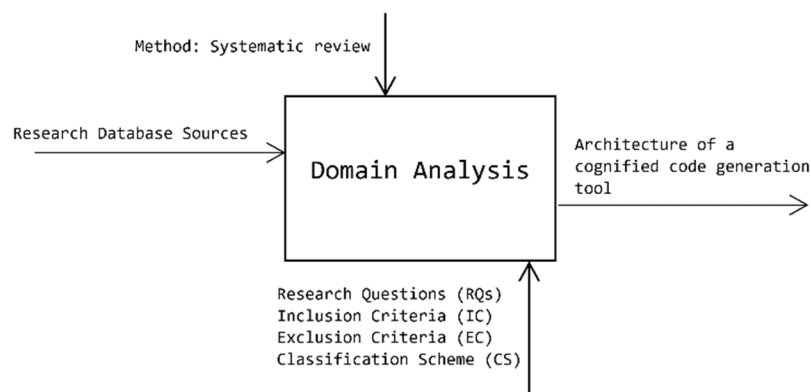
The resulting alternatives are demonstrated and documented using several feature diagrams.

#### 4.2. Definition of Research Questions

As one of the main goals of this work is not only adding recent investigations to give clarity to the domain of program synthesis overall, but it also aims to explore possible design choices and search for technique alternatives related to the four core dimensions of synthesis approaches. Eight research questions (RQs) that guide the domain analysis and contribute to the proposal of a conceptual design were selected after analyzing briefly related works and previous reviews.

The output of the analysis is provided at the end to readers as an architectural design of the promising cognified MDE code generation tool. Figure 1 demonstrates the model of domain analysis adopted in this work. These questions are addressed were as follows:

- RQ1: What was reported in peer-reviewed literature about paradigms and approaches to program synthesis with respect to the Software Engineering domain between 2003 and 2019?
- RQ2: What are the main characteristics (features) of each program synthesis approach reported in peer-reviewed research literature?
- RQ3: What are the common alternatives and trends in describing the user intent?
- RQ4: What are the common alternatives for describing the program space?
- RQ5: What are the common alternative approaches and trends used to deal with the synthesis problem?
- RQ6: What are common trending techniques that are applied over the program space for solving synthesis problem?
- RQ7: What are the spectra and trends of the program synthesis applications?
- RQ8: What kinds of search strategy were used for each application of program synthesis between 2009 and 2019?

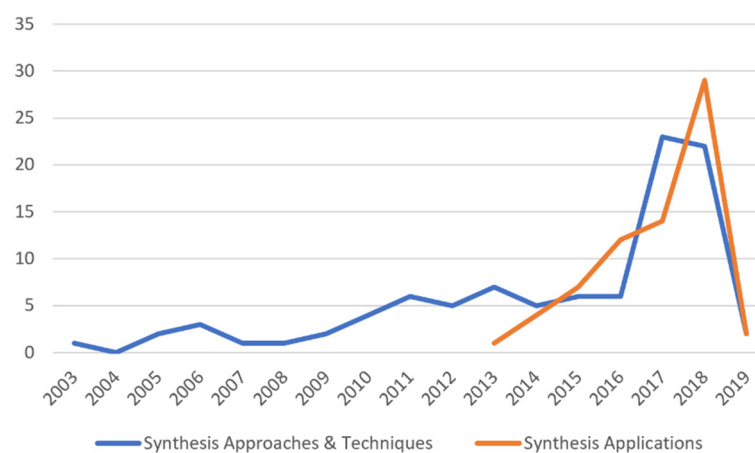


**Figure 1.** A summarized model of domain analysis.



### 4.3. Conduction of the Domain Analysis

A guided domain analysis of program synthesis was applied via a systematic literature review of the existing program synthesis approaches and frameworks from different research database sources available in the following globally recognized academic collections from 2003–2019: Scopus and ISI Web of Science, namely, IEEE Xplore, ACM Digital Library, ScienceDirect and Springer. The search covered two main directions (focuses): synthesis approaches and techniques (between 2003 and 2019) and their applications (between 2009 and 2019). This was accomplished to ensure the identification of a wider range of relevant research publications and proceedings that discuss program synthesis topics as well as their applications in formal software development methods. As it clearly seen in Figure 2, despite the broad time range chosen for this review and analysis, the number of considered publications noticeably increased in the most recent years, especially from 2009, reflecting of the active and more interesting research in the software engineering domain.



**Figure 2.** Range of selected publications.

The conducted search strategy was based on technical keywords with various search strings related to the domain of program synthesis, its paradigms, approaches, and applications. The search was executed in the selected research databases using the strings and keywords. The retrieved papers were inspected manually, one-by-one to check whether or not they were relevant to the research focus. Table 1 provides the detailed terms, as utilized in the search process. It is worth mentioning that the total number of selected publications was 182 papers before considering the RQs, IC, EC, and CS.

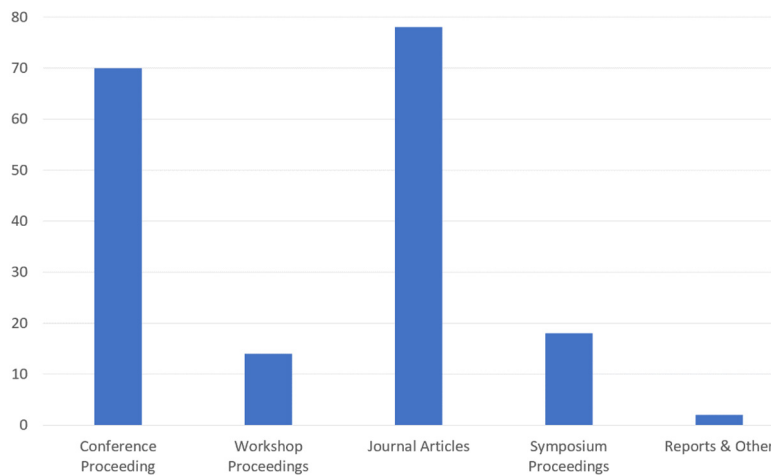
**Table 1.** Search Strings and Keywords.

Search Focus	Strings and Keywords
Program Synthesis Approaches and Techniques	(code OR program) AND synthesis (inductive OR deductive) AND (code OR program) AND synthesis Example-based AND (programming OR coding) (programming OR coding) AND by (examples OR demonstration OR sketching) (program OR code) AND (synthesis from examples) (Syntax OR Semantics OR Symbolic) AND-based AND (code OR Program) Synthesis
Applications of program synthesis	(program OR code) AND (suggestion OR completion OR repair OR correction OR recommendation OR comment OR documentation OR summarization OR generation OR transformation OR translation)

### 4.4. Inclusion and Exclusion Criteria

All resulting publications from the search strategy were initially reviewed, and only three categories of proceedings were considered: *conferences*, *workshops*, and *symposiums*, besides *journal*

*articles* and *reports*. From the 182 papers, only 12 papers that were either abstracts, positions, tool demos, posters, or non-English articles were excluded. In addition, two papers that were published before 2009 and related to the applications of program synthesis were eliminated, and four papers that were published before 2003 and related to the approaches and techniques of program synthesis were also eliminated, as illustrated previously in Figure 2. The following figure (Figure 3) shows the types of research publications included in the domain analysis.



**Figure 3.** Types of research articles included in the domain analysis.

#### 4.5. Classification Scheme

According to our research questions and inclusion and exclusion criteria, the 170 selected publications were classified, initially, based on their main topics, into categories belonging to three aspects. These aspects were paradigms of program synthesis, features of program synthesis approaches (techniques), and applications of program synthesis (Table 2).

**Table 2.** Classification of Publications.

Aspect	Categories
Paradigms of Program synthesis	inductive paradigm, deductive paradigm
Features/Techniques of Program Synthesis	user intent specifications, program search space, search technique
Applications of program synthesis	program repair, program summarization, program transformation (including code generation), program documentation and code completion

For the features of program synthesis aspect, the same key dimensions were used to present the categories of program synthesis. It is worth mentioning that the classification schema was developed and changed after starting the data extraction from selected articles and papers. This led to grouping and splitting of some categories. The final classification schema for this work is illustrated in Table 2.

#### 4.6. Data Extraction Strategy

Table 3 summarizes the information about synthesis techniques and their application domains collected from the extracted publications with respect to the defined RQs, IC, EC, and categorization scheme (CS).

**Table 3.** Extracted data with respect to the research questions (RQs), inclusion criteria (IC), exclusion criteria (EC), and categorization scheme.

Group ID	Reference No.	Techniques	Aspects
1	[15–23]	Recurrent Neural Networks (RNNs) and (deep) reinforcement learning with Natural Language Processing (NLP)-based user intent	Features and Techniques
2	[24–28]	Different sketching synthesis techniques for expressing user intent with annotated programs, e.g., execution-driven SAT/SMT-based and domain-specific rule sketching. User intent is expressed using a program sketch with some holes.	
3	[29–33]	Deductive reasoning, solver-based and programming-by-example with user intent expressed using domain specific languages (DSLs).	
4	[34–48]	Debugging information, programming-by-example with input/output (I/O) examples for expressing user intent.	
5	[49]	Counterexamples with oracle-guided inductive synthesis technique.	
6	[50,51]	These publications use logic formulas, the symbolic logic technique, weighted (tree) and context-free grammar for expressing user intent with SAT/SMT solving technique, statistical model.	
7	[52–54]	Constraints expressed using attribute grammar and refinement trees with RNNs, SMT-based solver	
8	[55]	Symbolic execution and extended finite-state-machine	
9	[37,56–60]	Execution traces with (RNNs), 2D drawing with convolutional neural networks (CNNs) and images with reinforced adversarial learning technique	
10	[61–64]	Execution traces for expressing user intent with different synthesis techniques such as the version space algebra and stochastic synthesizing	
11	[65]	Verification approach to solve the synthesis problem, specifications of atomic operations are used as input/output intent. Abstract finite tree automata used for expressing an initial program.	
12	[66,67]	Logic with approximation approaches to solve the synthesis problem.	
13	[68–73]	Optimization approaches are used to solve the synthesis problem.	
14	[33,48,64–79]	Constraint solving approach, SAT/SMT-based techniques to solve the synthesis problem.	
15	[80–83]	Neural networks, deep learning, machine learning and its related techniques to solve the synthesis problem.	
16	[84–89]	Template-based technique to express the program space.	Paradigms
17	[90–102]	Paradigms of program synthesis.	
18	[84,103–115]	Code completion and suggestion.	
19	[116–125]	Code summarization.	Applications
20	[126–135]	Code documentation and comments generation.	
21	[136–147]	Code compilation, generation, translation and transformation.	
22	[148–173]	Domain of code repair and correction.	

## 5. Paradigms of Program Synthesis

Since the 1960s, the problem of constructing the executable code of a desired program from higher-level descriptions has been considered. Two main synthesis paradigms have been distinguished over time, namely the inductive approach and the deductive approach. The inductive approach aims to derive the final program from some traces at a high level of specification, whereas the deductive approach aims to construct the final program from a type of specification that expresses a relationship between the input and output of a desired program [91,96,97,99,100,174].

The deductive program synthesis paradigm represents the large umbrella of the classic old school synthesis approach that many synthesis frameworks, over time, come under. Researchers who are interested in formal software engineering methods, where various activities are based on deductive reasoning (reasoning from the general to the specific), support the automation of development and processes, such as generating code from domain-specific languages or unified modeling language (UML) diagrams (transform a given low-level platform-specific model into executable code), as well as code optimization, theorem proving, model checking (proving that a given intermediate or low-level model meets a given specification), code static analysis, test case generation (producing automatic inputs that cause a code to fail to meet its high-level specifications), code verification (proving that a given code meets a given specification), code transformation (transforming a given code into an equivalent and more efficient one), logic programming (executing a code written in logic), and more [96,174–177].

On the other hand, the inductive program synthesis paradigm denotes another wider scope, which includes a variety of synthesis frameworks that have been emerging further with the rise of machine learning and artificial intelligence techniques during the most recent decades. These kinds of synthesis systems are based on inductive reasoning (reasoning from the specific to the general). They work, statistically, with a specific amount of data from a problem, such as I/O examples, test cases, computation traces of code, and desirable/undesirable behavior of a code to identify general patterns in data, and then they generate more complete programs [91,96,100].

In order to answer the first research question (RQ1) adequately, first, we had to distinguish between the inductive approaches and the deductive ones that fall within the scope of the inductive synthesis paradigm and deductive synthesis paradigm, respectively. This was achieved by applying the first level of domain analysis to research papers belonging to all subcategories associated with the first aspect. By going back to Table 3, it was found that from the 12 selected research papers that four papers were grouped under the deductive paradigm of program synthesis and discussed various approaches, whereas the rest of the publications (eight papers) were grouped as publications of an inductive paradigm of program synthesis and its related approaches (Figure 4).

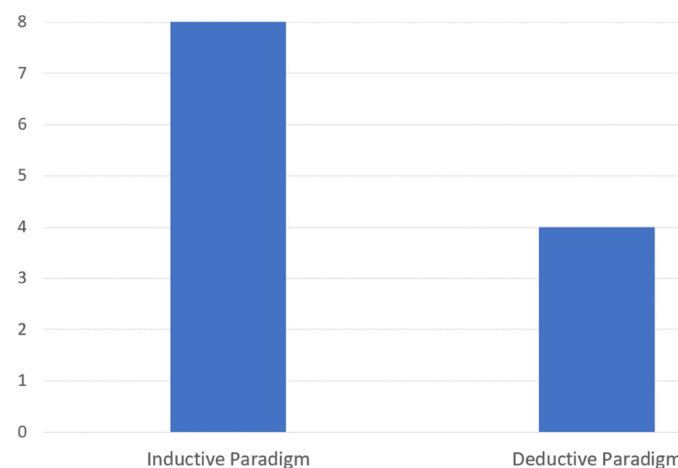


Figure 4. Distribution of publications on program synthesis paradigms.

The detailed results of the domain analysis in this part of the survey are documented using a feature diagram (Figure 5). The diagram illustrates all possible inductive-based and deductive-based program synthesis approaches covered in the domain analysis. The following subsections describe, in brief, the synthesis paradigm features illustrated in Figure 5.

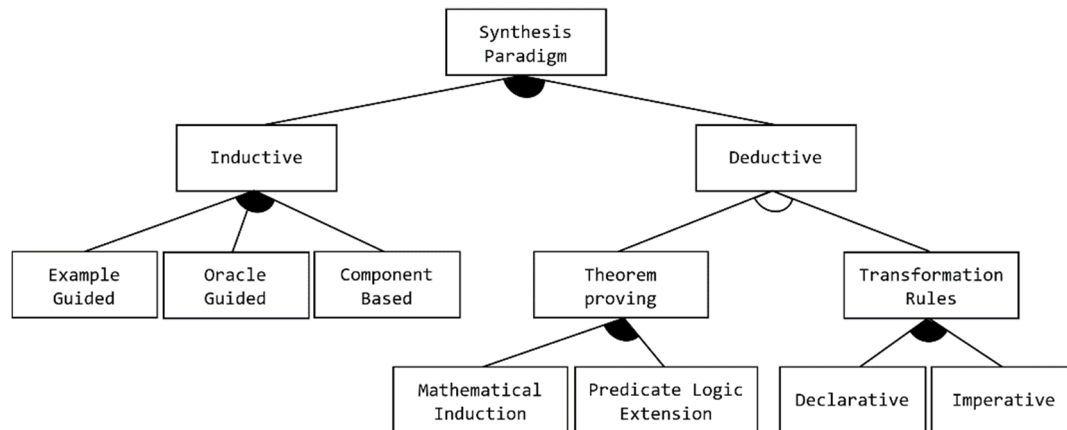


Figure 5. Program Synthesis Paradigms.

### 5.1. Inductive Program Synthesis Paradigm

This feature groups some characteristics for distinguishing approaches that follow the inductive paradigm of synthesis. As mentioned earlier, synthesizers that fall under this paradigm learn, statistically, from different forms of data, such as I/O examples, test cases, or code traces using a suitable searching algorithm or a machine learning (ML) technique to obtain final programs. The learner is considered a critical part of the design of synthesizers in inductive synthesis frameworks. The details of possible search techniques and learning methods, as well as the types of examples that might be given for the synthesis system are discussed and documented in Section 6.

Synthesis frameworks, based on the type of data provided for learning, have their own characteristics that might be used to classify inductive synthesis frameworks. This is documented in the lower layers of the feature diagram illustrated in Figure 5.

- **Example-Guided:** Unlike synthesis approaches that completely rely on deductive techniques to assemble user-intended programs, here, learning synthesizers from a small number of examples are required.
- **Oracle-Guided:** Frameworks under this subcategory use a querying system (Oracle) as part of the synthesizer design. They focus on the Oracle to answer (interactively) queries produced by a learner, such as the counterexample-guided approaches discussed in [95].
- **Component-based:** The process of assembling programs (loop-free) avoids the use of formal specifications and replaces them with a collection of existing functions/methods, composed to provide building blocks that are required for obtaining implementation detail. The component functions are interchangeable with a set of library functions provided by an application program interface (API) [92,93]. User intent can be described via a set of input/output examples, for example in [94], or even as a set of test cases, for example, in the work presented in [92]. In addition, the FrAngel approach [178] is able to synthesize loops and other control structures using a desired signature.

### 5.2. Deductive Program Synthesis Paradigm

This groups some subfeatures used in distinguishing approaches that follow the deductive paradigm of program synthesis. The classified frameworks under this paradigm allows a user to provide clear statements to specify a program without describing how they want to implement it.

A proof system is then used to produce the final code from the given specifications. There are many formal proof methods that can be adopted in a program synthesis framework that is based on the deductive paradigm. Each method has its own characteristics that might be used to classify synthesis frameworks. This is documented using the optional operator in the lower layers of Figure 5.

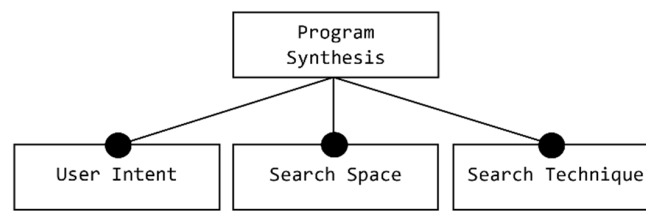
- **Theorem-proving:** This subfeature groups the deductive approaches that consider the derivation process of the final programs from specification as a problem of proving a mathematical or logical theorem. The given specification normally describes the relation between input and output without explaining the recommended way of implementing and computing it. It is worth mentioning that when trying to construct a program with recursive or iterative loops, the process of applying theorem-proving becomes more complex [172]. In this technique, for any input object of the program, the existence of an output that meets conditions is proved formally by one or more theories, such as:
  - *Mathematical Induction:* A proof method that is based on the principle of mathematical induction.
  - *Predicate Logic Extension:* A formal proof method based on logical theories.
- **Transformation Rules:** This subfeature groups the deductive approaches that rely on the direct application of transformations or program rewriting rules to a specification of a desired program [172]. The program derivation process, in this instance, is not regarded as a process of proving a theorem but as a process of transformational steps [171]. There are three mechanisms for expressing rules of transformation [30,135], as follows:
  - *Declarative Rules:* Each transformation rule is designed as a relation between the source and target without going into operational detail of how the relation can be achieved. The rules can be implemented using transformation languages such as Query/View/Transformation (QVT) Relations.
  - *Imperative Rules:* Each transformation rule is specified and designed as a number of operational mapping steps, which are required to obtain the target from the source, showing how the transformation itself is performed. The rules can be implemented using any Object-Oriented Programming (OOP) language, such as Java or C++.
  - *Hybrid Rules:* A combination of both declarative and imperative rules, which is represented in Figure 5. It is mandatory that (*at least one*) notation follows the features of transformation rules.

## 6. Features of Program Synthesis

This section aims to answer research questions RQ2, RQ3, RQ4, and RQ5. There are three main perspectives that must be considered if we want to investigate, classify, analyze, design, or construct a program synthesis system, namely, users (developers, programmers), programs of interest, and search techniques [101]. Dealing with these perspectives helps researchers and developers to draw a comprehensive view of program synthesis frameworks and systems. In previous works presented in [101,102,135], three dimensions of program synthesis that tackle these perspectives were discussed, namely, user intent, search space, and search technique. In the following subsections, these dimensions are reintroduced as top-level features that reflect critical points in the variation of program synthesis systems in the feature model (Figure 6).

After applying the second level of the domain analysis during the literature survey, results were documented using some feature diagrams, which are presented later in the following subsections. The following subsections explain and discuss features of the program synthesis approaches demonstrated in Figure 6. The mandatory notation is used in the above FD as every program synthesis system must consist of the three subfeatures to generate the final code from the high-level user intent specifications.



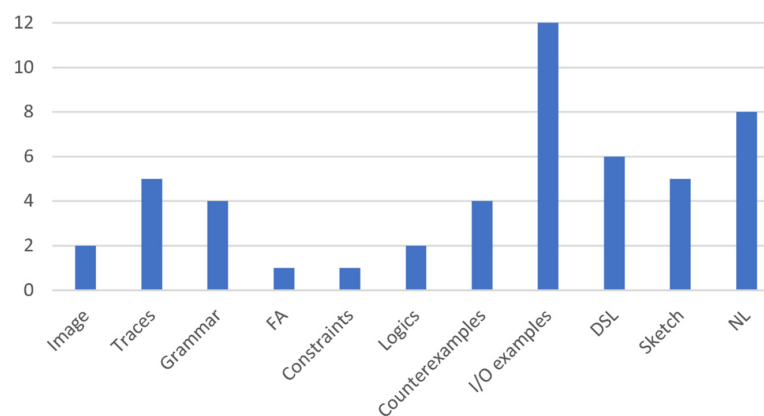


**Figure 6.** Features of program synthesis approaches.

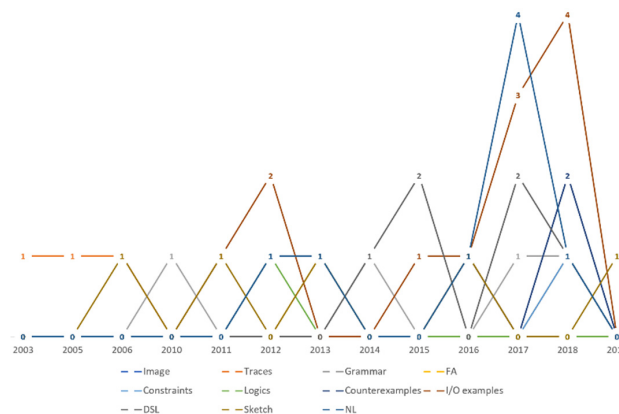
### 6.1. Developer & User Intent Specifications

Describing intent, or the specifications on the desired program [102], is the first significant dimension that is related to users or developers. When applying the adopted strategy of data extraction, which is based on the proposed RQs, IC, EC and Classification Scheme, it was found that the total number of papers that are grouped together under the Features and Techniques category with a focus on user intent was 50. Publications belonging to references groups: 1, 2, 3, 4, 5, 6, 7, 8, 9 and 10 are included in Table 3.

According to the domain analysis conducted on these groups of papers to start answering RQ2, there are different ways of expressing the user intent adopted in the various program synthesis approaches, such as input/output examples, formal specifications, logical relations or formulas, demonstrations, test cases, partial programs, (restricted) natural languages, and traces [101,102]. Figure 7 shows the distribution of publications on the various methods for expressing user intent. After reviewing the selected 50 papers and evaluating the publication dates and the methods of expressing user intent, the RQ3 was completely answered and the changes in expressing user intent trends between 2003 and 2019 are summarized in Figure 8.



**Figure 7.** Distribution of publications on the various methods for expressing user intent.



**Figure 8.** Changes in expressing user intent trends between 2003 and 2019.

Common features of these forms of user intent were derived after reviewing each paper; the mechanisms for describing user intent were named and determined during the second level of domain analysis as syntax-based, semantics-based, symbolic-based, and example-based mechanisms. The following feature diagram (Figure 9) illustrates the mechanisms used to describe the user intent.

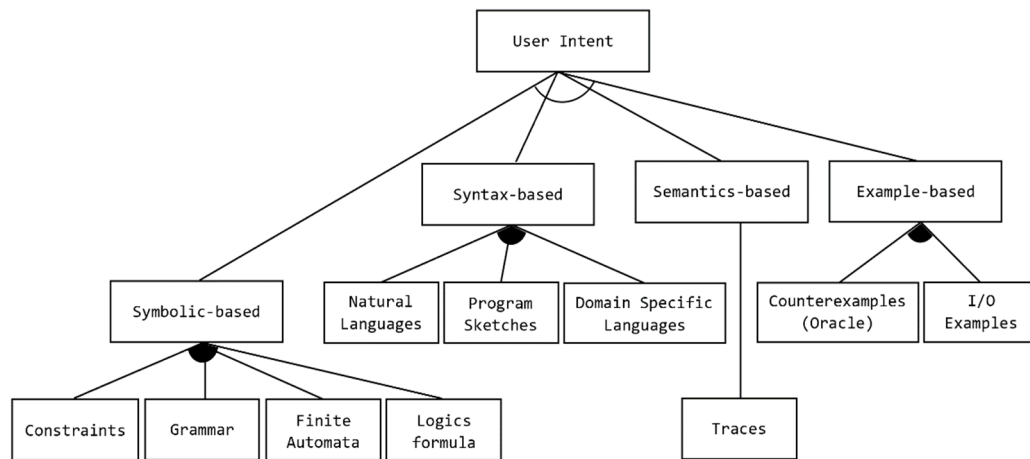


Figure 9. Features of user intent specifications.

Frameworks that fall under the syntax-based subfeature focus on a different format of syntax provided to the synthesis framework as user intent to produce the target code [179]. A mandatory (at least one) notation is used here to indicate that any combination of subfeatures is possible. The following points highlight, compare, and distinguish program synthesis approaches, briefly, in terms of the subfeatures illustrated in Figure 9.

- **Natural Languages:** Synthesis frameworks that fall under this subcategory use natural languages to describe the user intent program. Synthesizers then work to produce a program code from the NL description using a learning algorithm such as reinforcement or maximum marginal likelihood. The approaches presented in [116,139], and [19] are examples of synthesis frameworks that use an NL description for expressing user intent. Additionally, other frameworks, such as Tellina, adopt Recurrent Neural Networks to translate a program described using a natural language into an executable program [15].
- **Program Sketches:** According to frameworks presented in [25], synthesis frameworks allow a user to write an incomplete program (a program with holes or missing details) and a synthesizer and then derive the low-level implementation detail from the sketches by filling all given holes based on previously specified assertions.
- **Domain-Specific Languages:** A Domain-Specific Language (DSL) is a restricted set of a programming language that is designed to be understood and adopted for a particular domain. Similar to the structure of general-purpose programming languages (like C++ and Java), a DSL is a set of typed and annotated symbol definitions that form the DSL terminology [97]. These symbols can be either terminals or non-terminals that are defined using some high-level specification rules (e.g., context-free grammar). Each rule describes the transformation of every non-terminal into another non-terminal or terminal token of the language. All possible transformation operators and source symbols (tokens) are typed and located on the right-hand-side of the rules. Every symbol in the grammar is annotated with a corresponding output. The PROgram Synthesis using examples (PROSE) approach [31] is an example of a program that falls under the deductive synthesis paradigm (explained previously in Section 5) where the synthesis problem is solved using transformation rules and version space algebra such as FlashExtract [44] and FlashMeta [32]. Additionally, the solver-aided DSL (Rosette) that is based on theorem proving technique is designed in another DSL based approach for solving synthesis problem [33].

In addition, there are some synthesis frameworks that focus on learning their synthesizers using different forms of examples given as user intent, instead of providing a syntactic representation of the desired code. These frameworks are classified under the example-based category (Figure 9). The types of examples can be either one or a combination of I/O examples and counterexamples, or even traces. Programming by examples, as done in [16,39,41,44,47], is a common approach where a user expresses a desired code behavior using a set of I/O example pairs, and the synthesis tool constructs an executable implementation from these examples.

- **Input/Output (I/O) Example:** Frameworks under this subcategory adopt the use of I/O examples as an alternative strategy of expressing the user intent for a desired program. This kind of synthesis approach provides an interactive interface between the user and the synthesizer that allows the user to provide input/output example pairs until the desired program is reached, such as the approaches provided in [41,44,47].
- **Counterexample:** Synthesizers in frameworks under this subfeature adopt the so-called “Counterexample-guided” inductive synthesis strategy to produce possible candidate implementations from concrete examples of program behavior, whether this behavior is correct or not [27]. The synthesizer, in this instance, acts as a verifier or an Oracle in some approaches like [94,95] that performs a validation process on the candidate implementation code and produces (generates) counterexamples from its context to be used in the following iteration as input fed to the synthesizer. The counterexamples in this mechanism are used, iteratively, instead of new knowledge-free I/O examples generated for each solving iteration [27].

Moreover, frameworks that fall under the symbolic-based (computational) approach treat the program synthesis as a computational problem. Constraints, logic formulas, finite-state-machines, and context-free grammar are examples of symbolic notations that might be used to solve various computational problems in computer science. They can be adopted in the symbolic-guided synthesis framework as a representation of the synthesis problem to obtain the target code.

- **Logic Formulas:** The use of logical formulas is considered to be one of the classic methods for expressing high-level specifications of programs. There are two kinds of specifications considered for describing programs: semantics specifications and syntactic specifications. Frameworks that follow these subcategories use logic to describe semantic specifications, whereas they use grammar (e.g., context-free grammar) to describe constraints of syntactic specifications. Together, grammar and syntactic constraints provide a comprehensive template for the desired program. Using a template benefits the synthesizer by reducing the program search space [50,85].
- **Constraints:** Approaches that fall under this subfeature use formal language such as context-free grammar (attribute grammar) as a language for describing rich, structured constraints over desired programs. This kind of synthesis approach tries to tackle the problem of learning synthesizers, a rich set of constraints that must be satisfied from provided data, which is considered a difficult mission. The work presented in [54] is an example of this kind of synthesis framework.
- **Finite Automata (FA):** Frameworks under this subcategory allow a user to describe the desired program partially using finite state machines (FSMs) or, in some approaches, Extended FSMs with execution specifications and invariants to construct an FSM skeleton of the program. The synthesizer then completes the FSM skeleton from the desired specifications and invariants (supplied with the skeleton) using an inference technique [55]. In a TRANSIT tool [55], for instance, a computational-guided synthesis approach is adopted for a reactive system, where each process is expressed as an Extended FSM. The description of processes consists of a collection of internal state variables and control states and the transitions between them. The synthesis approach works by specifying these transitions through a set of guard conditions and an update code by inferring expressions using symbolic forms of functions, variables, and examples (Concolic snippets) to achieve a consistent system behavior.

- Grammar:** In addition, there are other frameworks that may involve program synthesis activities, such as program analysis and debugging. They use semantics information about a program, such as a bug report or memory address, instead of using actual program syntax or formal representations of it (e.g., grammar). These frameworks are categorized as under the semantics-based subcategory, demonstrated in Figure 9. It is commonly known that execution traces of programs consist of rich semantics information about the code. Thus, the use of execution traces has become widely accepted in the domains of program analysis and synthesis, which has brought remarkable results [57]. According to [57], many learning processes of (learning-based) synthesizers [58,59,152], are improved when using execution traces generated from I/O graphical image examples. Based on this idea, the approach presented in [57] uses execution traces that contain no control flow constructs as the specifications of a desired program along with I/O examples to train the proposed (neural) program synthesis model. As a result, the accuracy is improved to 81.3% from the 77.12% of their prior work. Frameworks that fall under the semantics-based subfeature, shown in Figure 8 above, use execution traces that contain semantics information about I/O values rather than using the I/O values themselves.
- Traces:** Frameworks with this subfeature provide a set of execution traces for learning synthesizers instead of a collection of I/O examples or logic rules [62]. This is because execution traces have been widely used for program analysis [63,64], where the traces are given as input to identify detailed (technical) characteristics about a program. Trace information may contain significant detail about the program, including dependencies, control flows (paths), values, memory addresses, and the inter-relationship between them. Reverse engineering techniques and tools are used to analyze traces and understand all possible scenarios and dynamic behaviors related to the code [64].

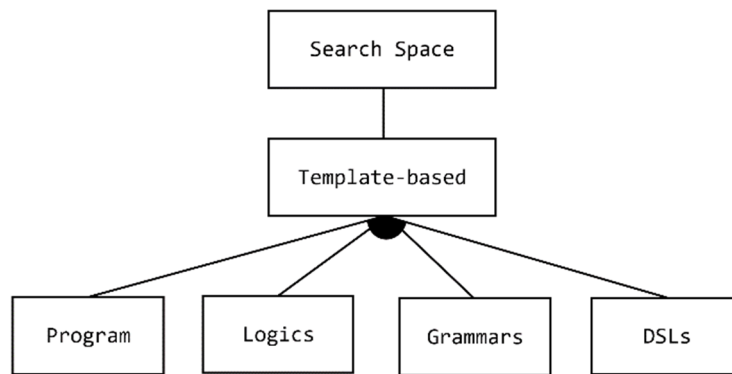
## 6.2. Search Space of the Program

The search space of a program is considered to be the domain of programs over which the desired program will be searched. Expressiveness and efficiency are two significant characteristics that must be considered by search space developers when designing the search space. On one hand, the expressiveness of the space should be adequate to describe all programs that users require. On the other hand, the space should be designed with a good degree of restrictiveness to allow it to perform an efficient search [148]. Reaching this balance between expressiveness and efficiency allows developers to create a good code synthesizer.

When applying the adopted strategy of data extraction, it was found that the total number of papers grouped under the program search space category was six. This is illustrated as group 16 (Table 3). According to the analysis conducted on these groups to continue answering RQ4, there is a variety of ways in which the search space can be expressed, for example, as a subset of an existing programming language, domain-specific language, context-free grammar, deterministic/non-deterministic FA, or logics [84,148].

During the second level of systematic domain analysis on the search space, it was found that templates are widely used across almost all kinds of program synthesis approaches [85,87–89]. Templates are considered to be a common technique that enables developers (users) to provide high-level insights about target programs to a synthesis framework using a generic programming or meta-programming feature (technique) available in some programming languages, such as C++, to create a template of a desired program. Template-based synthesis approaches can reduce the search problem and optimize the solving performance. The detail of possible types of solvers is covered later in other sections where the synthesis frameworks are categorized based on different adopted search strategies [88]. The creation of templates using programming languages or even formal specification languages (e.g., Z, Petri Net or Abstract Syntax Tree (AST)) or logic is considered a critical and difficult task, as the solver needs to translate the template back into an appropriate form for performing formal reasoning, such as logics or grammar [88] and then produce the complete target code.

After reviewing the selected six papers, it was found that the search space of a program can be expressed using four alternatives, namely, programming languages, logic, grammar, and domain specific languages. From that, RQ4 was completely answered. At the completion of this level of the domain analysis on the search space, the results were documented using a feature diagram (Figure 10). Some possible language combinations may appear to form the final search space template. That is why the mandatory (at least one) notation is used in Figure 10.



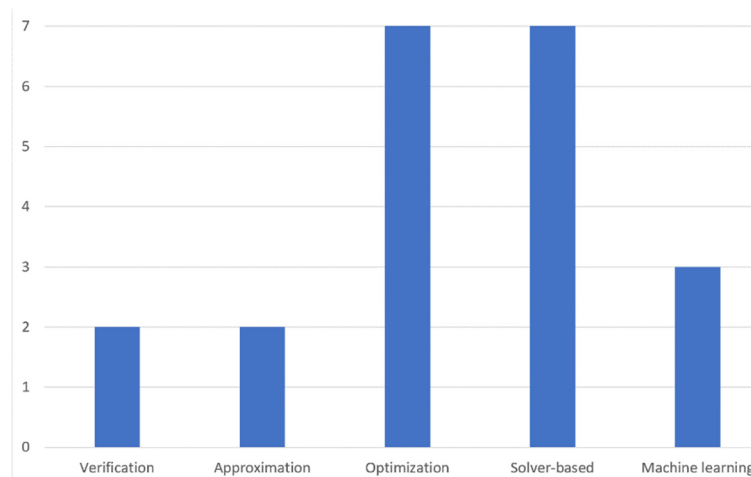
**Figure 10.** Features of Search Space of Programs.

### 6.3. Search Strategy

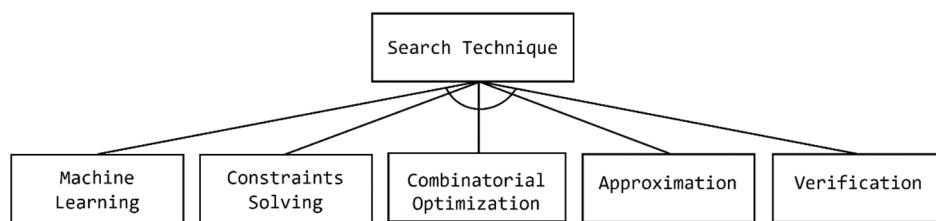
As mentioned earlier, the program synthesis problem is defined as a problem of finding an executable program that satisfies some high-level specifications and constraints. The process of searching over a program space to solve this problem is considered one of the three critical dimensions of any program synthesis approach. There are various search techniques and algorithms that might be adopted when designing code synthesizers based on whether the user intent specification is expressed via examples, partial program code, example pairs, or formal specifications [84,148].

This section is used to answer both RQ5 and RQ6. To answer RQ5 first, the data extraction strategy was applied, in which all publications belonging to reference groups 11, 12, 13, 14, and 15 were included (Table 3). The total number of publications considered under the search strategy category at this step was 21. These papers were evaluated based on the methods adopted for dealing with the synthesis problem and its variations over the period between 2005 and the middle of 2019. It was found that the program synthesis problem is tackled and treated from different perspectives as five kinds of computational problem, namely, the verification problem, the constraints satisfaction (solving) problem, the machine approximation problem, the combinatorial optimization problem, and the learning (statistical) problem (Figure 11).

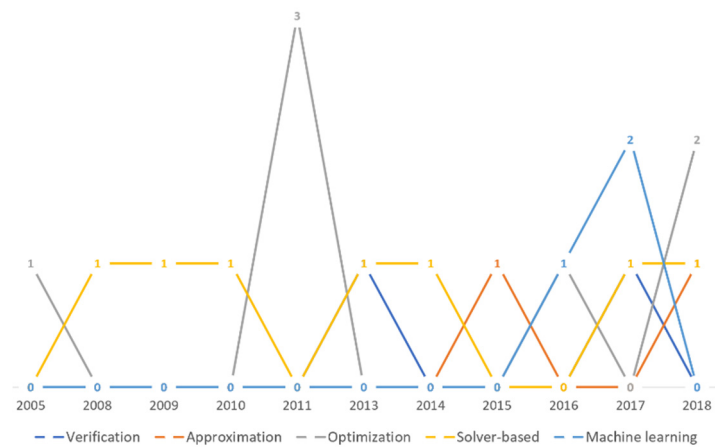
The alternative searching techniques used for solving the synthesis problem are documented in a feature diagram demonstrated in Figure 12. Additionally, in order to highlight the changes in this issue, Figure 13 summarizes the changes in handling the synthesis problem between 2005 and the middle of 2019. A remarkable increase in adopting machine learning (ML) and its related techniques and optimization techniques as search techniques for solving synthesis problems can be observed. Secondly, in order to answer RQ6, again the data extraction strategy was applied once more, in which all publications belonging to reference groups 11, 12, 13, 14, 15, 18, 19, 20, 21, and 22 were included (Table 3).



**Figure 11.** Distribution of papers using the various methods for dealing with synthesis problem.



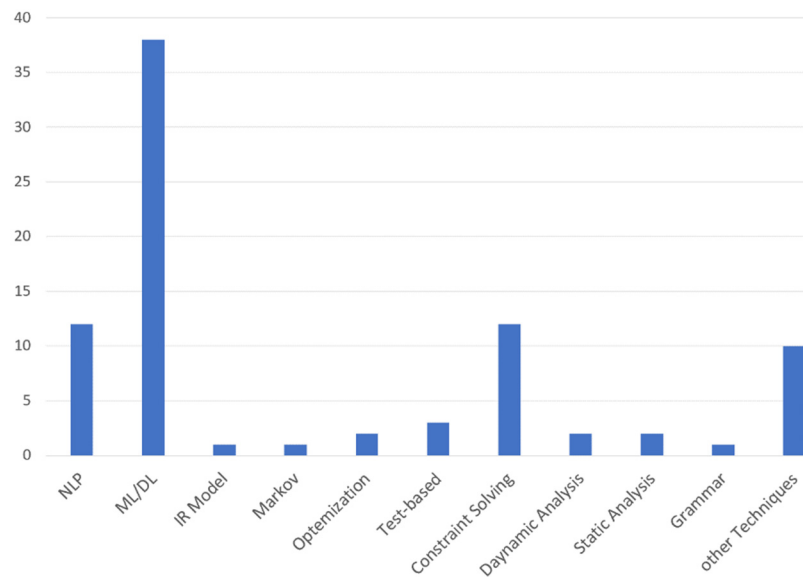
**Figure 12.** Features of Search Techniques.



**Figure 13.** Distribution of publications on the methods for dealing with program synthesis problem.

The total number of publications considered under the search strategy category at this step was 95, after eliminating some publications that did not mainly cover search techniques. These papers were evaluated based on the adopted technique used for solving the synthesis problem. Figure 14 demonstrates the distribution of publications on the solving techniques used for searching the program space to solve the synthesis problem. It is worth mentioning that the findings represented in Figure 14 were also used to answer RQ8, as described in the following section (Section 7). The following subsections compare and distinguish program synthesis approaches based on the features of the search technique, as demonstrated in the above top-level feature diagram (Figure 12).





**Figure 14.** Distribution of publications on the solving techniques used for solving synthesis problem.

- Verification:** Verification can be defined as a process of solving a problem by checking that a program satisfies a high-level specification on all inputs brought from a very large or infinite set. Program synthesis frameworks that fall under the verification subcategory encode the synthesis problem as a verification problem to be solved using certain verification tools. This kind of program synthesis follows the correct-by-construction philosophy from the old school program design approach that appeared in the 1970s and 1980s. In the program verification approach, as discussed in [60], code statements are encoded as logical facts with some guards to be examined.

The tool then infers some invariants and program statements until the program is synthesized automatically. The proof of correctness for these generated conditions is then produced theoretically using reasoning techniques. According to [65], the second phase of the common counterexample-guided inductive program synthesis approach is based on iterative verification processes. A verification step is performed on the candidate program in order to discover a counterexample input that violates the specification. This process continues until the candidate program checking is accomplished by either passing the verification check or failing a synthesis check [65].

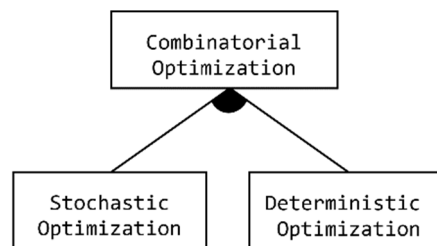
- Approximation:** Traditional program synthesis frameworks generally produce programs that only meet specifications without the guarantee that they will be the optimal solution. Some synthesis approaches treat the program synthesis problem as an approximation problem. The approximation problem usually results in the discovery of approximate solutions (the nearest solution) to the optimal one.

Synthesis approaches in this category aim to automatically produce optimal programs that approximately meet a desired correctness specification with certain attributes, for example, the fastest program [66,67]. According to [67], a collection of large problems, where candidate programs have a search space that is too large and hard to explore, is introduced as a PARROT benchmark suite. The sketch-based program synthesis approach is used to solve PARROT problems, resulting in more efficient program solutions with a reasonable level of accuracy for all seven problems. It is worth mentioning that the syntax-based synthesis framework (SyGuS) fails to solve any PARROT collection problem [67].

- Combinatorial Optimization:** Combinatorial optimization is a type of mathematical optimization that reflects, in general terms, the process of selecting the best value that satisfies some given criterion from some available alternatives [74]. Combinatorial optimization is considered to be a field of theoretical computer science that solves discrete optimization problems through

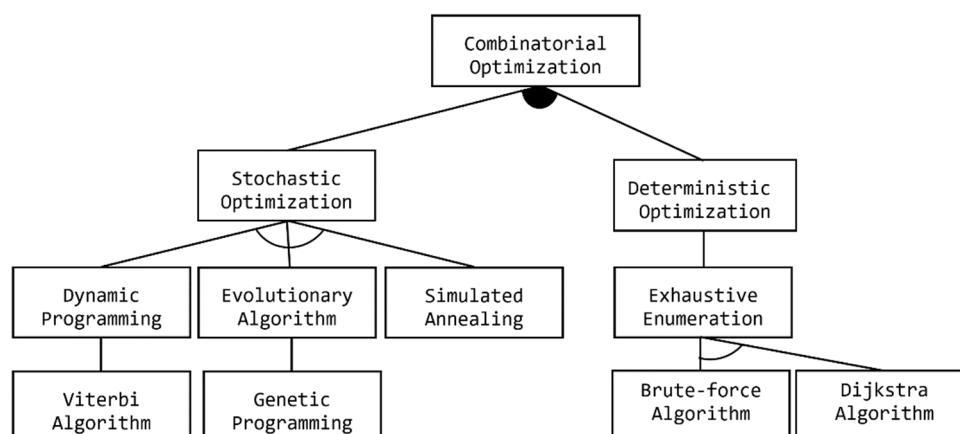
finding an optimal solution from a finite set of possibilities [74]. In the optimization process, an original program  $\mathcal{P}$  is given, and a search technique is applied over the program space to find another functionally equivalent program  $\mathcal{Q}$  that satisfies some constraints for performance enhancement [69]. It improves the search performance by reducing the size of the search space. Although various real-world problems, such as finding variable assignments that satisfy constraints, partitioning graphs, coloring graphs, and more can be solved numerically by combinatorial optimization, most of these problems are often subject to uncertainty [74].

This caused the emergence of two widely adopted resolution methods of combinatorial optimization, namely, Stochastic Optimization and Deterministic Optimization (Figure 15). In some approaches, both techniques may be used together. This is illustrated in the FD as a mandatory (at least one) subfeature.



**Figure 15.** Top-level Features of combinatorial optimization in synthesis.

- **Stochastic Optimization:** The Stochastic Optimization method involves solving combinatorial optimization problems that involve uncertainties, whereas the deterministic one focuses on finding solutions for combinatorial optimization problems by evaluating a finite set of discrete variables. For each method, several efficient algorithms have been designed and successful search techniques have been adopted for solving many real-world problems, including program synthesis (demonstrated in the detailed feature diagram in Figure 16). According to the domain analysis, the Evolutionary Algorithm (Genetic programming), Dynamic Programming Algorithm (e.g., Viterbi algorithm), and Simulated Annealing are stochastic algorithms that are used in several program synthesis frameworks as techniques for seeking the target code constructs from the high-level specifications [68,155].



**Figure 16.** Detailed Features of Combinatorial Optimization in Program Synthesis.

- **Dynamic Programming:** Dynamic Programming is an optimization technique that simplifies complex problems by boiling them down into many overlapping subproblems. Solutions of these simpler subproblems are combined to provide an optimal solution to the complicated problem. In a nested problem structure, a relation between the value of the larger problem

and the values of the subproblems is specified, and each computed value of a subproblem's solution is used recursively to find the overall optimal solution to the problem [70].

It is worth mentioning that the synthesis problem must be described in a way in which its solution is constructed from some solutions to overlap subproblems [71]. Many implementation algorithms that improve the overall performance of the optimization process are based on dynamic programming, such as the divide-and conquer [70] and linear-time dynamic programming algorithms presented in [71].

- *Simulated Annealing*: The principles of simulated annealing were inspired and inherited from physical properties in annealing solid mechanics. In physics, defects of solids are removed first by heating the solids up to a high temperature and then transforming them into crystal materials by a slow cooling process. At the highest temperature, the material is considered to be at the highest (max) energy state, whereas the minimum energy state is the frozen state [72].

Simulated annealing was introduced into the domain of computer science as a probabilistic strategy for solving combinatorial optimization problems with a large search space. For example, simulated annealing is used, via the Real-Time Software System Generator (RT-Syn) framework [72], to minimize the related resource costs of software applications, including design and maintenance, by synthesizing the implementation detail of the design. When considering program synthesis as an optimization problem, some crucial implementation decisions must be involved during problem resolution, such as data structures, control flows, and algorithms. In simulated annealing, the program space is treated as a configuration space that encompasses all legal decisions.

Iteratively, a random current feasible design with some perturbations (move set) is proposed. At the end, this move set must achieve all feasible designs in the design space. In each iteration, a cost function is used to measure the goodness of the current design in order to find the best design that can be reached. The last characteristic of the simulation is the cooling schedule, which mimics the cooling process of materials in physics. Moves in the high-energy state that decrease gradually in the cost function are accepted to produce a suboptimal solution, whereas a quick decrease results in a near-optimal solution to the problem [72].

- *Evolutionary Algorithm*: An evolutionary algorithm is a kind of generic population-based optimization that is inspired by biological evolution mechanisms, such as reproduction, mutation, recombination, and selection. In biology, biological changes in characteristics, or evolution, occur when evolutionary mechanisms and genetic recombination react to these changes, resulting in different characteristics becoming more common or hidden in the population in the following generations [96,169]. In the domain of computer science, In the domain of computer science, algorithms for solving optimization problem is applied to a population of individuals, where fitness functions are used iteratively over the population to evolve the quality of the final solution [73].
- *Genetic Programming (GP)*: is a kind of evolutionary algorithm that uses genetic operations, namely, mutation, crossover, and selection to evolve its populations iteratively, until the best solutions to a given optimization problem are achieved [154]. It performs better than the exhaustive search when searching a problem (program) space that is too broad, because the search over the space is guided by the measures produced by the fitness function [96,169]. GP is considered one of the common techniques that is applied in the domain of program synthesis and automatic program repairs [73,96,154,155,169].
- *Deterministic Optimization*: On the other hand, the simpler alternative to the Stochastic method is the Deterministic Optimization method, which is used in some synthesis frameworks as an implemented search technique (Figure 16). Exhaustive Enumeration is considered to be a very general search-based problem-solving technique that involves all possible alternatives

to be examined during the problem resolution process in order to find the optimal solution to the problem [68]. The brute-force algorithm is considered to be a common technique of exhaustive enumeration optimization, as noted during the conducted domain analysis. In this optimization, three kinds of collection optimization input are given: formal representation of candidate expressions  $E$ , logical specification and constraints  $S$ , and a finite set of examples  $X$ . The targeted problem to be solved must satisfy the following (First-Order Predicate Logic with equality) formal rule:

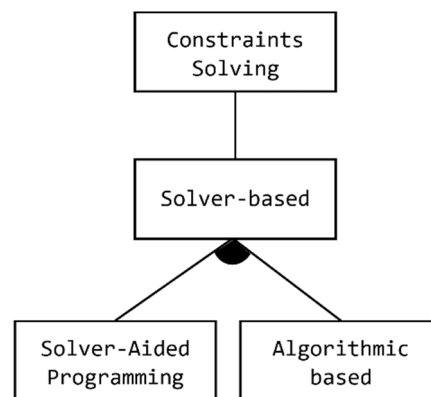
$$\forall (e \cdot (e \in E) \wedge s \cdot (s \in S) \wedge x \cdot (x \in X) \cdot \text{solve}(e, s, x)) \neq \text{FALSE} \quad (1)$$

However, the rapid (exponential) growth of the search space, which occurs due to the program's size or other reason, is considered a crucial problem that deterministic-based synthesizers may face, even when a powerful optimization technique is adopted. In order to solve this problem, the synthesis framework must be improved to guide the search using the weighted directed graph and decision tree in the approaches mentioned in [55,69], respectively. A probabilistic model is used as guidance for the search-based synthesizer.

The model takes a set of program tokens, including terminal and non-terminal ones, and produces a probability for each production rule. A weighted directed graph with a sentential form for each node and a calculated weight for each edge is then derived from the model. The enumeration search based on this improved structure decreases the search by considering the shortest path from the source node via graph search algorithms such as Dijkstra's algorithm [69].

- **Constraint Solving:** The theory behind constraint solving program synthesis begins by expressing the semantics of a given program in some logic formulas. Instead of compiling the program into such a low-level executable machine code, it is compiled into logical constraints (formulas) as an intermediate representation of the given program. A solver-based strategy is then applied via solver-aided verification or synthesis tools to solve the condition satisfaction problem through proofing the correctness of the given program. It tries to find an input that makes the program fail (if it exists) when such a constraint is unsatisfied in an automatically generated test. Here, the program synthesis problem is treated as the Constraints' Satisfaction Problem (CSP). The CSP can be defined as a collection of mathematical questions that are considered to be objects that must satisfy some constraints. Some intensive research has been conducted in the artificial intelligence (AI) and operational research domains when solving the CSP.

The feature diagram shown in Figure 17 classifies program synthesis frameworks with respect to those approaches that solve the synthesis problem as a CSP using theorem provers (logical reasoning techniques). The solving approach can be achieved by adopting either the Boolean Satisfiability Problem (SAT solver), the Satisfiability Modulo Theories (SMT solver), or a combination of both. A common solving strategy that is based on logical reasoning is aimed at reducing the second-order search problem to (first-order) constraint solving first. A type of solver (SAT or SMT) is then used to solve the constraint problem. The solving-based tool can be integrated within some program synthesis approaches like syntax-based synthesis, as discussed in [78]. This is expressed in the following FD by the mandatory (at least one) notation.



**Figure 17.** Features of constraints solving synthesis approaches.

Solvers can be implemented using two strategies, namely, solver-aided programming and an algorithmic based approach. In the algorithmic-based approach, the written implementation is often complex and hard to understand with an informal correctness proof. It is described normally using a high-level specification language (e.g., Hoare logic) supported by one theorem proving system (e.g., Isabelle [74]) to prove its correctness [79].

Another approach for implementing solvers is adopting an appropriate solver-aided domain specific language (e.g., Rosette [33,48]) and tools. DSLs are used to package the insights and knowledge of domain experts and allow other people who are interested in that domain application to effectively solve problems in that domain [30]. Rosette is a solver-aided DSL that is built on the top of a programmable programming language called Racket to enable the development of the kinds of tools that are based on program verification and synthesis concepts [33,48].

Unlike the algorithmic-based approach where a compiler must be built from a programming language into the constraint solving system, which is an extremely hard task, DSL simplifies the task by building these special kinds of compilers to build an interpreter for the DSL language, or just a library or an API when using an embedded type of DSL. The interpreter of the language requires a so-called symbolic virtual machine to translate the given program semantics into constraints. While using solver-aided DSL language, the synthesis framework becomes simpler and better, as the translation from the language into constraints is obtained automatically [33,48].

- *Boolean Satisfiability Problem (SAT solver):* The Boolean Satisfiability Problem (SAT) can be defined as a problem for checking whether or not a formula that is expressed using Boolean logic is satisfiable. SAT solving is considered the cornerstone of several software engineering applications, such as system design, model checking and hardware, debugging, pattern generation, and software verification [33]. The SAT problem is denoted as the first proven nondeterministic polynomial time (NP-complete problem) in which algorithms in their worst-case complexity that involve thousands of variables and millions of constraints are used for solving [33,75]. There are several program synthesis frameworks that use the SAT solver to resolve the synthesis problem, which is implemented based on an algorithmic approach using C++ or Python. For instance, the SKETCH framework utilizes the SAT solving technique in a counterexample-guided iteration that interacts with a verifier to check the candidate program against the specification and generates counterexamples until the final program that meets the complete specifications is found [27,94]. Additionally, SAT solving and the so-called gradient-based numerical optimization technique are combined and used for solving program synthesis problems in the Real Synthesis (REAS) framework [76]. The search space in REAS is explored using the SAT Solver for solving constraints on discrete variables to fix the set of Boolean expressions that appear in the program structure. This allows better tolerance with approximation errors, which leads to efficient approximation results. The REAS technique is implemented within the SKETCH framework. The end user, a programmer, writes their program with a set of unknowns using the high-level SKETCH

language to express the intent. These unknowns are Boolean expressions (constraints) that need to be solved [76].

- Satisfiability Modulo Theories (SMT solver): The Satisfiability Modulo Theories (SMT) is a technique that is used to find satisfying solutions for the First-Order Logic (FOL) with an equality formula. The FOL formulas include the Boolean operations, belonging to Boolean Logic, which have more complicated expressions than variables including functions, predicates, and constants, as sometimes, the adoption of SAT solvers for a program synthesis problem requires richer logic formulas. Thus, in SMT formulas, some propositional variables in the SAT formula are replaced with some First-Order predicates. These predicates are Boolean functions that return the Boolean values of some variables [77]. The use of the Satisfiability Modulo Theory (SMT) solvers has emerged as a useful tool for verification, symbolic execution, theorem proving, and program synthesis approaches. There are many available SMT solvers, such as Z3 and the Cooperating Validity Checker (CVC4), that are used for solving the program synthesis problem. These frameworks are implemented based on an algorithmic approach using general-purposes programming languages [77].

For instance, Z3 is a new SMT solver that is implemented in C++ and produced by Microsoft Research to tackle software analysis and verification problems. It works as a reasoning engine that proves the correctness of programs or discovers their errors by analyzing the verification conditions. Additionally, Z3 acts as a test case generation tool in which it is used to produce new test cases with different behaviors from the execution traces of the program [77]. Some techniques appear to integrate SMT solving tools with various synthesis approaches. According to [78], the syntax-guided approach of program synthesis allows users to provide hints to guide the synthesizer to find solutions to its synthesis conjectures. Moreover, SMT solvers are used to solve synthesis conjectures. The CVC4 SMT solver is, as shown in the work presented in [78], extended with some capabilities to make it efficient for synthesis conjectures using two embedded techniques, namely, Quantifier Instantiation and Syntax-Guided Enumeration.

- Machine Learning: Machine learning (ML) is an application of the artificial intelligence (AI) branch of computer science that enables the machine to learn from a massive amount of data without being explicitly programmed. In the context of software engineering, ML techniques have brought great advances in program synthesis, in which they may be used to create automated tools with better code comprehension ability to help developers to understand and modify their code using knowledge extraction or recognition techniques [81]. Thus, the synthesis problem is introduced here as a machine learning problem. Developers who are interested in following this approach to solve the synthesis problem find themselves faced with a variety of independent choices, expressed in FD 17 by some optional notations. Different learning techniques are used to guide the synthesis search and automatically decompose the problem synthesis, such as deep learning, neural networks, reinforcement learning, and version space learning. These learning styles are illustrated in Figure 18.
- *Version Space Learning*: Version Space Learning is commonly used in programming-by demonstration (PBD) synthesis applications. In the PBD approach, a programmer demonstrates how to perform a task, and the system learns an appropriate representation of the procedure of that task. Version Space is considered to be a logical approach to machine learning where the concepts of learning are described using some logical language.



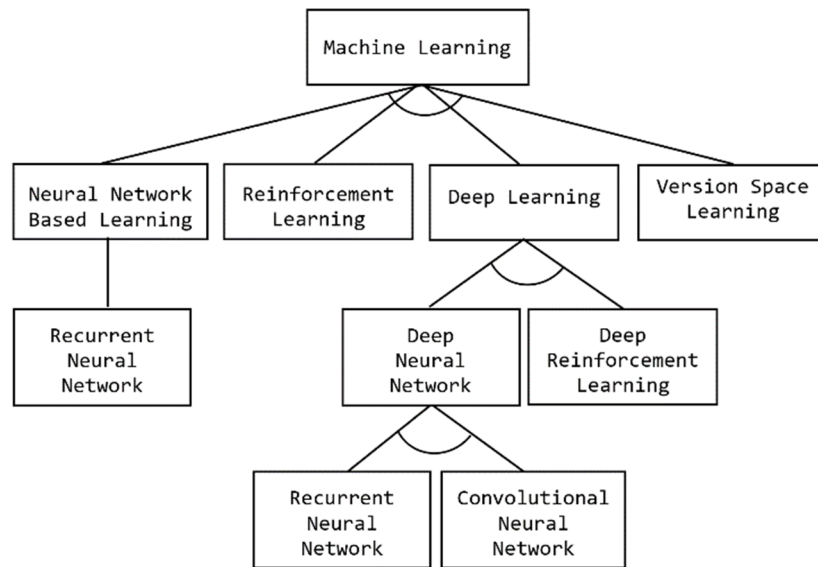


Figure 18. Features of machine learning based synthesis approaches.

The learning process can be seen as a search function over the space of the hypothesis that maps from a complex object into a binary classification. Different learning algorithms might be used to search over the space. This space is a hypothesis that is a set of disjunction logical formulas, which can be defined as

$$\forall H_m, H_n : \text{HYPOTHESIS} \cdot (H_m \vee H_n) \quad (2)$$

In this approach, the learning algorithm uses a number of training examples to restrict the space of the hypothesis. Each inconsistent hypothesis with a given example is removed from the space. This refinement process of the hypothesis space is called the candidate elimination algorithm.

$$\begin{aligned} \forall H_m, H_n : \text{HYPOTHESIS} \cdot (H_m \vee H_n) \wedge \\ \forall x : \text{EXAMPLE} \cdot (\text{satisfies}(x, H_m) \wedge \text{satisfies}(x, H_n)) \end{aligned} \quad (3)$$

According to [62], an extended version space with algebraic operations is used for learning the synthesis approach from execution traces of programs, in addition to the inputs and outputs of programs. Algebraic operators, such as the union, intersection, join, and transformation operators are used to construct complex version spaces. This allows an efficient, exhaustive search of the program's space that is consistent with the training execution traces. The designed learner is able to recognize the control structures of a program, such as IF and WHILE statements, as well as an array data structure. In the evaluation of this approach, it was shown to provide correct results from a small number of training examples.

Additionally, the SMARTedit framework was introduced as a (PBD) application based on version space algebra [180]. Reusable version space components have been designed beside version space algebra for the domain of text editing that supports a subset of Emacs command language. Any type of mapping between inputs and output, including Boolean values and structured objects, is considered.

- *Reinforcement Learning:* Reinforcement learning (RL) is considered to be a subfield of machine learning that aims to teach an agent how to perform a specific task and achieve a goal in an uncertain, potentially complex environment. Many RL applications have been emerging with the rapid advancement in the domain of games technology and robotics. In the context of program synthesis, reinforcement learning algorithms are applied within various frameworks to maximize the likelihood of generating semantically correct programs, as well as to tackle program aliasing issues when different programs may satisfy a given specification [19,139].

It is obvious that there is a limitation in other supervised machine learning techniques when dealing with program aliasing, for example. With the synthesis approach presented in [139], the process of generating any consistent program with the given I/O examples is directly encouraged by using policy gradient reinforcement learning instead of only optimizing the maximum likelihood. Furthermore, a syntax checker is used to prune the space of possible programs, which helps to generate better programs.

Not only this, but reinforcement learning has been presented in a transformation approach from natural language mapping into an executable program, such as the approach presented in [19]; reinforcement learning (RL) has been integrated with the maximum marginal likelihood (MML) paradigm. This resulted in a new learning algorithm that can be applied to a neural semantic parser and showed significant results. It can deal with spurious program bias by adopting an exploration strategy that is based on approximating the policy gradients of both LR and MML, which guide the exploration task [19].

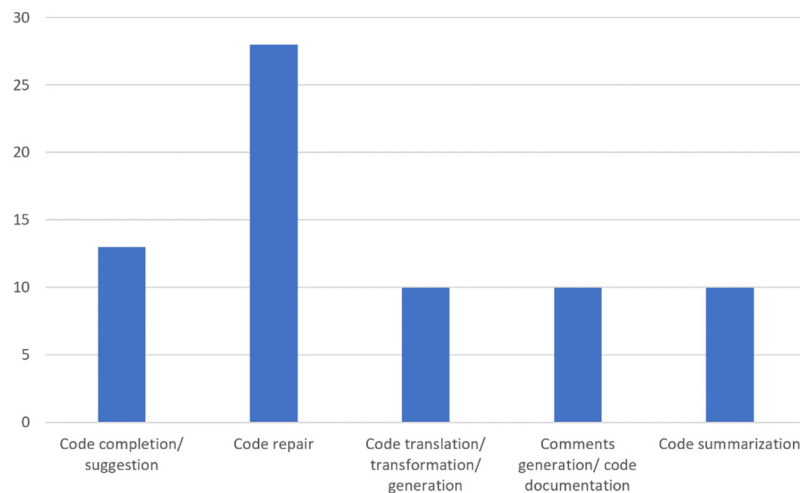
- *Neural-Network-Based Learning:* A neural network can be defined as an interconnected group of artificial neurons that use a mathematical or computational model for information processing. They are used to solve AI problems through building classification and prediction systems to make predictions. According to the domain analysis, neural-network-based approaches to program synthesis have gained greater attention from the software engineering research community. This is reflected in the popularity of NNs for machine learning in recent years. Several recent research works have introduced neural-network-based frameworks and approaches to program synthesis from I/O examples [57].
- *Deep Learning:* As mentioned earlier in this paper, deep learning (DL) can be defined as a branch of machine learning where the architecture of a learning approach consists of multiple layers of data processing units. There is a variety of synthesis frameworks that adopt deep learning techniques, such as deep neural networks (Convolutional and Recurrent NNs) and deep reinforcement learning [116]. The RobustFill framework [82], for instance, is a neural program synthesis framework based on RNN that allows variable-length sets of input/output examples (pairs) to be encoded.
- *Domain Specific Language (DSL)* is used in RobustFill to express the collection of transformation rules of different textual operations, such as substring extractions, constant strings, and text conversions. The adopted DSL has the ability to express complex textual expressions (strings) by employing an effective regular expression extraction technique. The DSL takes a given string as the input and returns another string as the output. The synthesis system is trained with a number of I/O examples and has been shown to achieve 92% accuracy. It is worth mentioning that during the conducted domain analysis, we found various deep learning techniques adopted in different program synthesis applications, such as DeepCom [126] and CRAIC [127] for code comment, the CDE-Model [118] for code summarization, DeepRepair [156] for code repair, and RobustFill [82] and DLPaper2Code [145] for code translation and generation.

Further comparison and detail of these frameworks is beyond the scope of this paper. For each mentioned framework, only the kind of DL technique adopted was extracted to be used in developing the lower level of the above-mentioned feature diagram (Figure 18).

## 7. Applications of Program Synthesis

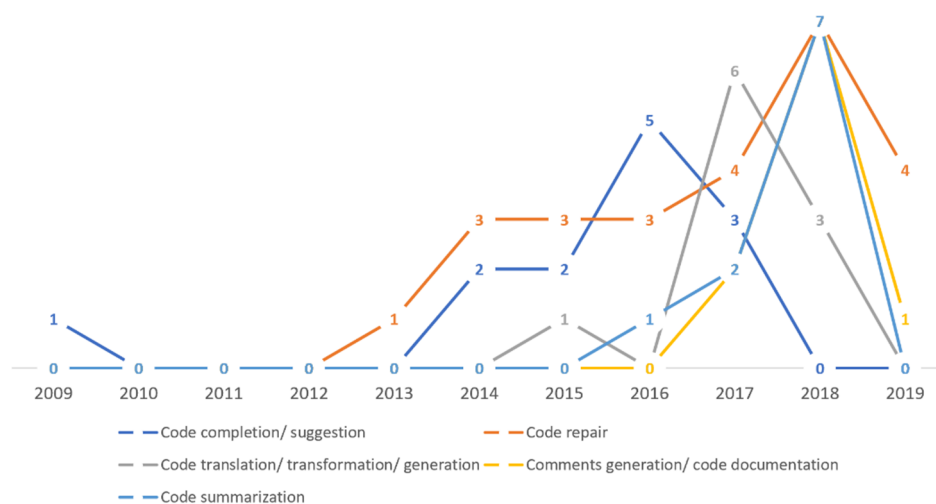
This section aims to answer RQ7 and RQ8 through a concise revision and evaluation of a number of publications that discuss the applications of program synthesis from the software engineering domain perspective. When applying the data extraction technique based on the RQs, IC, EC, and Classification Scheme, the total number of papers grouped together under the applications of program synthesis category was 71. All publications belonged to reference groups 18, 19, 20, 21, and 22 (Table 3).

According to the analysis conducted on these groups of publications to answer RQ7, there are various modern applications built on top of synthesis frameworks, namely, code completion, code repair, code suggestion, code transformation, code summarization, code documentation, and code generation. The following figure (Figure 19) shows the distribution of papers on the different modern program synthesis applications.



**Figure 19.** Distribution of papers on applications of program synthesis.

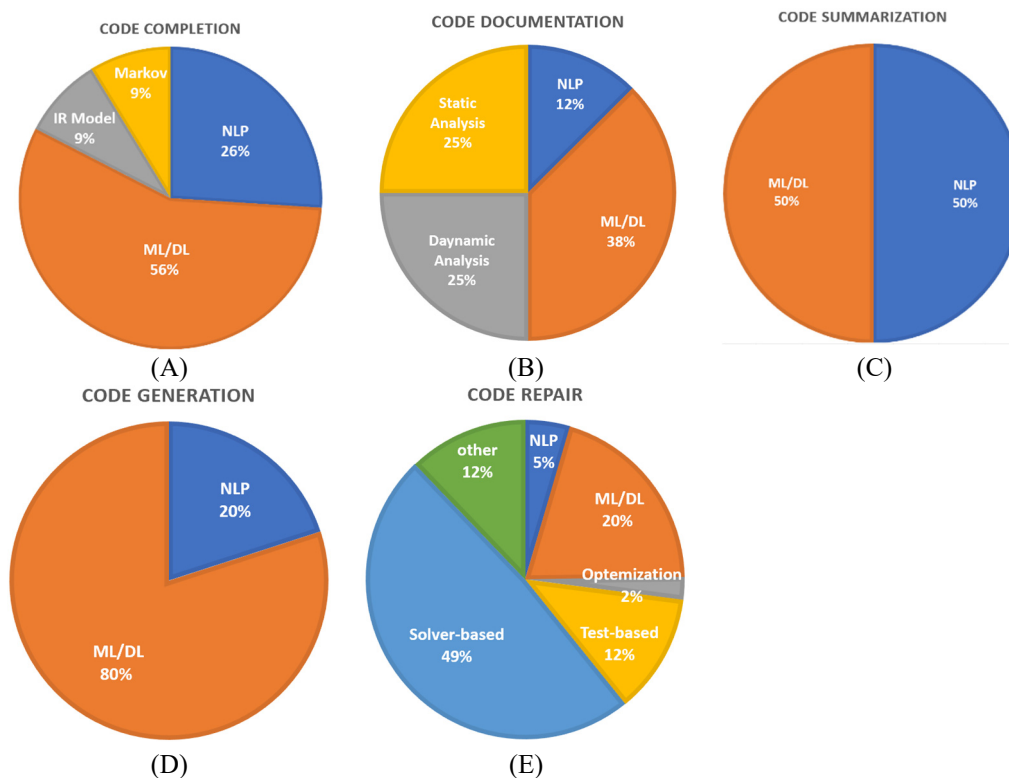
In order to completely answer RQ7, changes in the program synthesis applications between 2013 and 2019 are expressed in Figure 20. Remarkable growth in the number of publications related to code documentation, code repair, and code summarization can be observed in the last two years regarding contrast code completion and code transformations/generation.



**Figure 20.** Changes to program synthesis applications between 2009 and 2019.

Answers to RQ8 were provided from the interesting findings of RQ6 and RQ7. Various (ML/DL) techniques, such as Neural Networks (NNs), Recurrent Neural Networks (RNNs), Artificial Recurrent Neural Networks (LSTM) and Reinforcement learning have been used in various publications to solve the program synthesis problem in the covered applications with the highest number of selected papers, 27 out of 71 (38%). The percentages of studies related to each application category were as follows: 80% for code transformation/generation, 50% for code summarization papers, 50% for code completion, 30% for code documentation and only 17.8% for code repair papers.

In addition to this, it was also found that different Natural Languages Processing (NLP) techniques, such as n-gram and pattern-based (nano and micro) techniques, have been used in various publications to solve the synthesis problem in all categories of applications, representing the second highest number of selected papers with 11 out of 71 (15.5%). The most popular types were code summarization papers (with 50%) and code completion papers (with 23%). Figure 21 demonstrates the distribution of program synthesis application.



**Figure 21.** Distribution of program synthesis application. (A) Demonstrates the percentage of synthesis techniques adopted in the various code completion applications covered in the analysis. (B) Represents the percentage of synthesis techniques adopted in code documentation applications covered in the analysis. (C) Shows the percentage of the two common code summarization techniques covered in the domain analysis. (D) Demonstrates the two main techniques used in code generation application based on program synthesis covered in the analysis. (E) Illustrates the percentage of various techniques used in the applications of code repair.

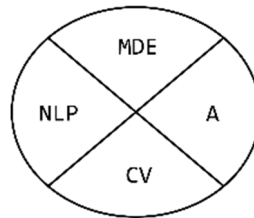
## 8. Architectural Design of a Suggested Code Generation Framework

The resulting features of program synthesis approaches from the conducted analysis are discussed in this section by highlighting the future direction for the design of a code generation framework that is based on the cognified code synthesis phase. It is commonly known that the traditional code generation phase focuses on generating a final executable code from platform-specific (design) models (PSMs) using efficient approaches and techniques. This is unlike the code synthesis stage, which aims to produce an executable code from higher specifications (user intent) using sophisticated search techniques and descriptive high-level user intent specifications.

In the proposed work, it is recommended that a learning-based synthesis step, which supports both machine learning and NLP, is considered in the design of the proposed code generation framework to make it cognified. The following subsections characterize the recommended features of a synthesis tier and explain the architectural design, conceptually, without going into depth on the implementation detail or design choices, for example, how to implement the chosen machine learning and NLP algorithms to solve the synthesis problem.

### 8.1. Concepts

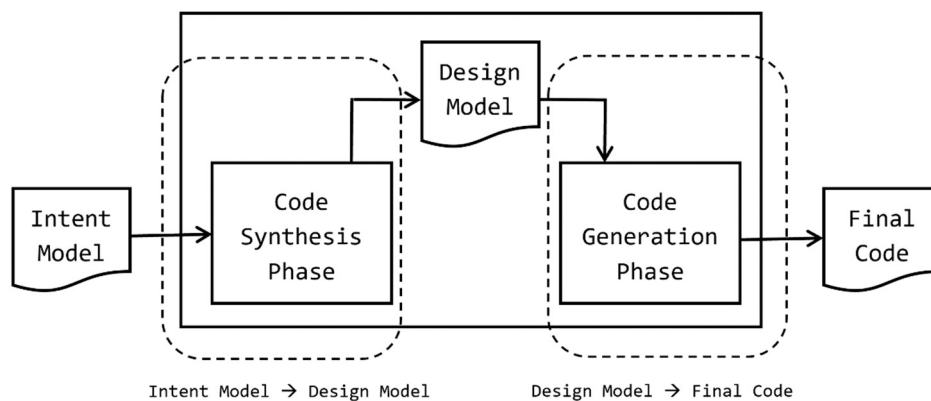
The suggested cognified code synthesis tier can be described as a combination of four subfields of computer science, namely, model-driven engineering (MDE), natural languages processing (NLP), computer vision (CV), and artificial intelligence (AI), as illustrated in Figure 22.



**Figure 22.** Concepts used in the proposed synthesis tier.

### 8.2. Architecture of the Synthesis Engine

Figure 23 demonstrates the overall architectural design of the cognified generation framework. It shows that there are two main transformation phases, namely, the code synthesis phase and the code generation phase. Both phases are performed by the synthesis engine component. The code synthesis phase aims to produce a platform-specific design model from a higher and more abstract model expressed using non-technical domain expert knowledge (Intent Model), rather than producing the final executable code.

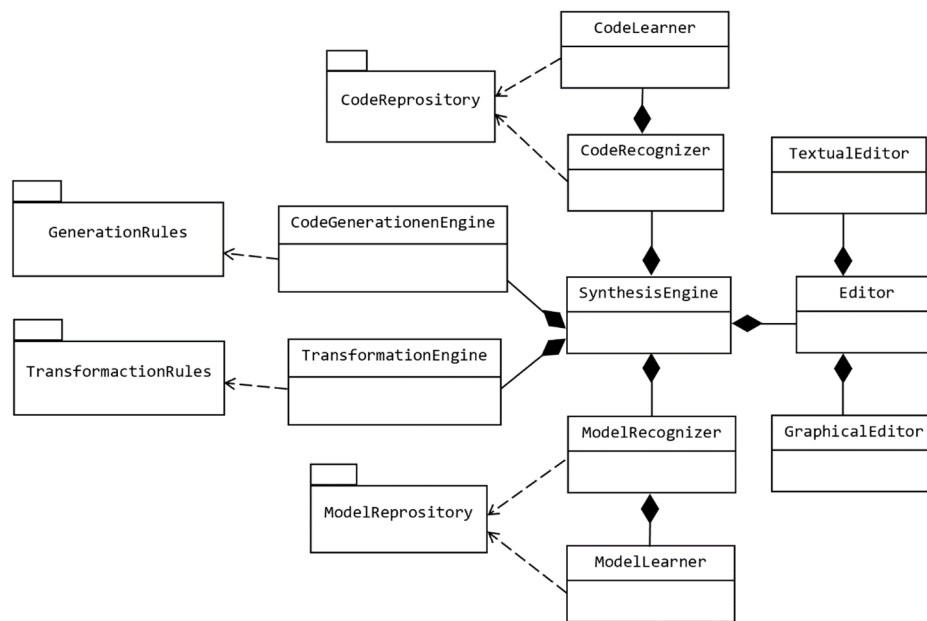


**Figure 23.** Overall architecture of the proposed framework.

There are four core components in the proposed synthesis engine (Figure 24) that are responsible for orchestrating and executing all required transformations at the synthesis phase to produce the design model. These components are the editor, code recognizer, model recognizer, and transformation engine. On the other hand, the code generation phase consists of a number of domain-specific generators (agents) that are responsible for generating the final executable code for a target environment. The key component that is responsible for performing this process is the code generation engine.

With respect to the variety of user intents covered in the proposed classification system (Section 6), here, the user intent is described using some hand-drawn models containing graphical and textual characteristics. The textual details (features) might be constraints, such as Object Constraint Language (OCL), or a textual DSL, whereas the graphical details reflect the actual graphical notation of some modelling languages, such as UML.

The visual and textual characteristics of models, or the inputs to the synthesis engine, are recognized by the synthesis system via two recognizers (Figure 24). The ModelRecognizer is used for detecting the graphical features of the hand-drawn models through an appropriate AI system (machine/deep learning) and computer vision techniques. Furthermore, the CodeRecognizer is used to capture the textual features on the models through AI and NLP techniques.



**Figure 24.** Transformations component of the proposed architecture.

### 8.3. Recommended Search Technique

According to the presented classification system (Section 6), the search techniques of the proposed synthesis engine can be categorized as machine learning-based search techniques. Many advanced deep neural networks, especially convolutional neural networks (CNNs) and recurrent learning systems that are connected to some huge model/neural networks (RNNs), have been used for code generation, as well as object detection and text analysis. In the proposed design, it is suggested that these possible search techniques are implemented via two recognizer components. Thus, the demand for the suggested recognizers to be supplied by models and code repositories for training purposes has emerged.

### 8.4. Language Model (Program Space)

It was suggested that the domain specific language (DSL) approach should be used in this work [97]. It is common for the abstract and concrete syntax and semantics of an external DSL or domain-specific modeling language (DSML) to be defined via the following various strategies: grammar-based (e.g., the Backus–Naur form (BNF)), metamodeling, and the UML profiling approach. The details of each strategy are outside the scope of this paper. The metamodeling approach was selected at this stage of development. Indeed, it is important to mention that the design language must be expressive enough to be able to capture various real-world domain features. At the same time, the language must also be restrictive enough to be able to describe the problem domain precisely, and it must be cognified, using structured examples for training, in the future.

Firstly, to represent the characteristics captured in the hand-drawn models, two levels of DSL that form intermediate representations of models were used, namely, the Intent model and the Design model. The user intent model was expressed via a DSL language that is closely related to the hand-drawn model domain, whereas the design model was described using low-level language that was close to the domain of the implementation platform but was platform-independent.

The UML class diagram (Figure 25) demonstrates a snapshot concept only at the metamodel level that was used to define the Intent model; however, a detailed description of the full version of this DSL along with its semantics is outside the scope of this work. This partial metamodel was used to exemplify the proposed model through a simple case that expresses a hand-drawn table. It is worth



mentioning that the following UML activity diagram (Figure 26) shows the data flow of models from the hand-drawn form into the final executable code.

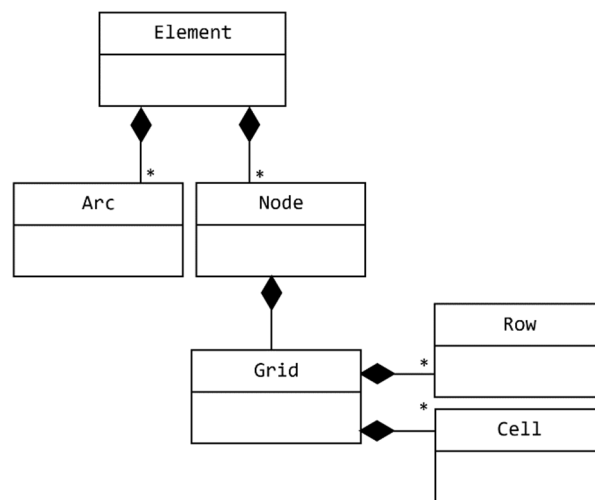


Figure 25. A snapshot of the Intent metamodel.

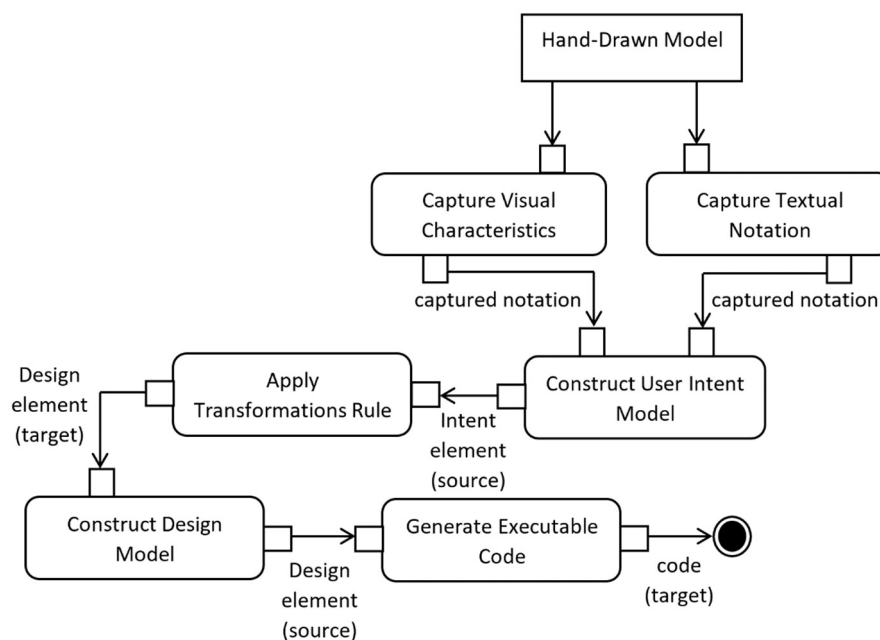


Figure 26. The process of model evolution to obtain the final executable code from hand-drawn models.

#### 8.4.1. Intent Model

This model was constructed as a result of solving vision problems to extract important graphical features from the hand-drawn model. Unsupervised feature learning was performed by using the CNN and RNN techniques to learn the recognizer. This example-based approach can be used to train the system, as many model and code repositories are emerging on the Internet nowadays. The pre-processing preparation of the data and training approach are outside the scope of this work. The following snapshot (Listing 1) is a possible Extensible Markup Language (XML) representation of the Intent model.

As seen in Listing 1, the language of the Intent model consists of graphical terms, such as grid and cell. Similarly, the recognizer must recognize other shapes like arrows, lines, circles, and more.

**Listing 1.** Intent model DSL language (XML format) for representing a hand-drawn grid shape.

```

<Grid no = "1">
  <Row no = "1">
    <Cell no = "1" text = "rigNo" underlined = "true" />
    <Cell no = "2" text = "name" />
    <Cell no = "3" text = "age" />
    <Cell no = "4" text = "contact" />
  </Row>
  <Row no = "2">
    <Cell no = "1" text = "70812324" />
    <Cell no = "2" text = "John Smith" />
    <Cell no = "3" text = "31" />
    <Cell no = "4" text = "John.Smith@gmail.com"/>
  </Row>
</Grid>

```

#### 8.4.2. Design Model

The model was constructed as the result of applying a set of model transformation rules by the transformation engine. It is considered a low-level model that is expressed using a language close to the implementation and code. The following snapshot (Listing 2) is a possible XML representation of the database design model. It is worth mentioning that the design model is platform-independent. From Listing 2, it can be seen the language of the design model contains various terms, such as table, column, record, primary key (PK), and field, taken from database systems.

**Listing 2.** Design model DSL language (XML format) for representing a transformed table structure.

```

<Table id = "1" name = "table1">
  <Header>
    <Column id = "1" label = "rigNo" PK = "true" null = "false" />
    <Column id = "2" label = "name" size="30" />
    <Column id = "3" label = "age" range="[18-60]" />
    <Column id = "4" label = "email" format="EMAIL" />
  </Header>
  <Record id = "1">
    <Field id = "1" value = "70812324" type = "Number" />
    <Field id = "2" value = "John Smith" type = "Text" />
    <Field no = "3" value = "31" />
    <Field id = "4" value = "John.Smith@gmail.com" type = "Text" />
  </Record>
</Table>

```

#### 8.5. Transformations Engine

As shown in Figure 24, the role of the (model) transformation engine component is clear. It transforms elements expressed in the user intent model into low-level elements of the platform-independent design model, at the code synthesis phase. This can be achieved by applying a set of transformation rules that represent a complete model transformational system. As mentioned earlier, a case of generating MySQL schema was utilized to exemplify the possible rules of transformations that might be considered in the proposed approach.

By following the strategy of scattering the whole transformation into a collection of transformational agents (composed transformation) as an internal architecture, as suggested in [181], several suggested transformational rules/agents for generating MySQL schema from hand-drawn sketches (models) can be listed:

- **Detecting Table:** This transformation agent is responsible for transforming the captured hand-drawn grid shapes from the Intent model into a platform-independent table structure.

This transformation step ensure that every transformed table has a unique name, even if the source grid has no name. The following First-Order Predicate Logic (FOPL) rule expresses the mapping between a grid element in the Intent model and a table element in the Design model.

$$\begin{aligned}
 & tr : Grid \rightarrow Table \\
 & \forall g : Grid, m : IModel. (g \in m) \\
 & \rightarrow \\
 & \exists t : Table, n : DModel. ((t \in n) \wedge NameOf(g, t))
 \end{aligned} \tag{4}$$

- **Detecting Column:** This transformation agent is responsible for transforming the captured hand-drawn cell of a grid from the Intent model into a platform-independent column structure. This transformation step includes deciding whether the captured column is a primary key or not. The mapping rule between a cell element and a column one can be expressed using the following logical formula as

$$\begin{aligned}
 & tr : Cell \rightarrow Field \\
 & \forall c : Cell, g : Grid. (c \in g) \\
 & \rightarrow \\
 & \exists t : Table, f : Field. ((f \in t) \wedge NameOf(c, f))
 \end{aligned} \tag{5}$$

- **Detecting Record Instance:** This transformation agent is responsible for transforming the captured hand-drawn rows of a grid from the Intent model into a platform-independent recorded instance. The mapping rule between row and recorded elements can be expressed using the following FOPL logical formula as

$$\begin{aligned}
 & tr : Row \rightarrow Record \\
 & \forall r : Row, g : Grid. (r \in g) \\
 & \rightarrow \\
 & \exists t : Table, d : Record. ((d \in t) \wedge NameOf(d, f))
 \end{aligned} \tag{6}$$

### 8.6. Code Generation Engine

The code generation engine is responsible for translating the platform-specific design model into executable code at the code generation phase. As demonstrated in Figure 24, the generation engine is a component of the proposed synthesis engine. Unlike the discussed components, the role of the code generation engine component is straightforward. By following the previously mentioned strategy of scattering the whole transformation into a collection of transformational agents that are structured as a hierarchal composition structure (Section 8.5), as suggested in [181], a collection of transformational agents (generators) was adopted in the design of the code generation engine. These platform specific agents are responsible for generating executable code in a target environment.

As mentioned earlier, a case of generating MySQL schema was utilized to exemplify the possible rules of transformations that might be considered in the proposed approach. Figure 27 shows the internal design of a code generator agent to generate MySQL schema from the design model.

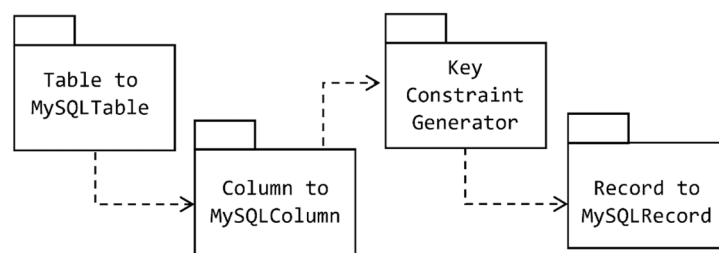


Figure 27. Hierarchical Composition of MySQL Code Generation.

## 9. Conclusions

Program synthesis is expanding rapidly. In this paper, a feature model for describing program synthesis approaches was introduced as a result of applying a systematic three-phase domain analysis to related publications on existing program synthesis paradigms and approaches and their applications. The selected 170 publications were considered and reviewed in the designed three-phase systematic domain analysis. The RQs, IC, EC, and classification scheme were identified and applied to the related publications, which resulted in the classification of synthesis paradigms into inductive and deductive paradigms. Synthesis approaches were also classified based on three main features, namely, user intent, program space, and search technique. All results were documented in a number of feature diagrams using the feature-based modelling technique.

Although there have been various successful deductive-based program synthesis approaches since the 1980s, many new frameworks have been proposed over the last five years for various applications, which are based on inductive-based approaches, especially machine (deep) learning and NLP. This is because of the accelerated wave of deep learning and computer vision advancement in the past few years. This recent transition has led to program synthesis gaining many benefits from the massive number of code samples and program traces available on different online repositories and datasets for solving the synthesis problem. This, in other words, contributes to the cognification of the program synthesis task as a part of a complete code generation framework.

In this respect, the results of the conducted analysis are considered to motivate, highlight, and give further insight regarding the architectural design of the promising cognified code generation framework that is supplied by a learning-based code synthesis engine. This is a contribution associated with one application of program synthesis, which is code generation. A synthesis engine is recommended and discussed as a tier of the proposed architecture to enable developers to express their intents using hand-drawn models and code hand-writing. The engine is able to capture critical features from both textual and graphical sketches of code and evolve them into the final executable code via a series of transformational steps.

**Funding:** This research received no external funding.

**Conflicts of Interest:** The author declares no conflict of interest.

## References

1. Visser, E. A survey of rewriting strategies in program transformation systems. *Electron. Notes Theor. Comput. Sci.* **2001**, *57*, 109–143. [\[CrossRef\]](#)
2. Cristina, D.; Pascal, K.; Daniel, K.; Matt, L. Program Synthesis for Program Analysis. *ACM Trans. Program. Lang. Syst.* **2018**, *40*, 45. [\[CrossRef\]](#)
3. Church, A. Logic, arithmetic and automata. In Proceedings of the International Congress of Mathematicians, Institut Mittag-Leffler, Djursholm, Sweden, 15–22 August 1962; Volume 1962, pp. 23–35.
4. Bodik, R.; Jobstmann, B. *Algorithmic Program Synthesis: Introduction*; Springer: Berlin/Heidelberg, Germany, 2013.
5. Buchi, J.R.; Landweber, L.H. *Solving Sequential Conditions by Finite State Strategies*; Springer: New York, NY, USA, 1967.

6. Rabin, M.O. Automata on infinite objects and Church's problem. *Am. Math. Soc.* **1972**, *13*, 6–24.
7. Summers, P.D. A methodology for LISP program construction from examples. *J. ACM* **1977**, *24*, 161–175. [[CrossRef](#)]
8. Pnueli, A. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (sfcs 1977), October 1977*; IEEE: Piscataway Township, NJ, USA, 1977; pp. 46–57.
9. Emerson, E.A.; Clarke, E.M. Using branching time temporal logic to synthesize synchronization skeletons. *Sci. Comput. Program.* **1982**, *2*, 241–266. [[CrossRef](#)]
10. Manna, Z.; Wolper, P. Synthesis of communicating processes from temporal logic specifications. In *Workshop on Logic of Programs*; Springer: Berlin/Heidelberg, Germany, 1981; pp. 253–281.
11. Manna, Z.; Pnueli, A. *The Temporal Logic of Reactive and Concurrent Systems: Specification*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2012.
12. Kupferman, O.; Vardi, M.Y.; Wolper, P. An automata-theoretic approach to branching-time model checking. *J. ACM* **2000**, *47*, 312–360. [[CrossRef](#)]
13. Mens, T.; Van Gorp, P. A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.* **2006**, *152*, 125–142. [[CrossRef](#)]
14. Czarnecki, K.; Helsen, S. Feature-based survey of model transformation approaches. *IBM Syst. J.* **2006**, *45*, 621–645. [[CrossRef](#)]
15. Lin, X.V.; Wang, C.; Pang, D.; Vu, K.; Ernst, M.D. *Program Synthesis from Natural Language Using Recurrent Neural Networks*; Tech. Rep. UW-CSE-17-03-01; University of Washington Department of Computer Science and Engineering: Seattle, WA, USA, 2017.
16. Manshadi, M.H.; Gildea, D.; Allen, J.F. Integrating programming by example and natural language programming. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, Washington, DC, USA, 14–18 July 2013.
17. Locascio, N.; Narasimhan, K.; DeLeon, E.; Kushman, N.; Barzilay, R. Neural generation of regular expressions from natural language with minimal domain knowledge. *arXiv* **2016**, arXiv:1608.03000.
18. Kushman, N.; Barzilay, R. Using semantic unification to generate regular expressions from natural language. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Atlanta, Georgia, 9–14 June 2013; pp. 826–836.
19. Guu, K.; Pasupat, P.; Liu, E.Z.; Liang, P. From language to programs: Bridging reinforcement learning and maximum marginal likelihood. *arXiv* **2017**, arXiv:1704.07926.
20. Krishnamurthy, J.; Mitchell, T.M. Weakly supervised training of semantic parsers. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, Jeju Island, Korea, 12–14 July 2012; pp. 754–765.
21. Rabinovich, M.; Stern, M.; Klein, D. Abstract syntax networks for code generation and semantic parsing. *arXiv* **2017**, arXiv:1704.07535.
22. Zhong, V.; Xiong, C.; Socher, R. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv* **2017**, arXiv:1709.00103.
23. Sun, Y.; Tang, D.; Duan, N.; Ji, J.; Cao, G.; Feng, X.; Zhou, M. Semantic parsing with syntax-and table-aware sql generation. *arXiv* **2018**, arXiv:1804.08338.
24. Murali, V.; Qi, L.; Chaudhuri, S.; Jermaine, C. Neural sketch learning for conditional program generation. *arXiv* **2017**, arXiv:1703.05698.
25. Samimi, H.; Rajan, K. Specification-based sketching with Sketch. In *Proceedings of the 13th Workshop on Formal Techniques for Java-Like Programs, July 2011*; ACM: New York, NY, USA, 2011; p. 3.
26. Solar-Lezama, A.; Tancau, L.; Bodik, R.; Seshia, S.; Saraswat, V. Combinatorial sketching for finite programs. *ACM Sigplan Notices* **2006**, *41*, 404–415. [[CrossRef](#)]
27. Solar-Lezama, A. Program sketching. *Int. J. Softw. Tools Technol. Transf.* **2013**, *15*, 475–495. [[CrossRef](#)]
28. Hua, J.; Zhang, Y.; Zhang, Y.; Khurshid, S. EdSketch: Execution-driven sketching for Java. *Int. J. Softw. Tools Technol. Transf.* **2019**, *21*, 249–265. [[CrossRef](#)]
29. Raza, M.; Gulwani, S.; Milic-Frayling, N. Compositional program synthesis from natural language and examples. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence*, Buenos Aires, Argentina, 25–31 July 2015.
30. Subahi, A.F.; Alotaibi, Y. A New Framework for Classifying Information Systems Modelling Languages. *JSW* **2018**, *13*, 18–42. [[CrossRef](#)]

31. Polozov, O.; Gulwani, S. *PROSE: Inductive Program Synthesis for the Mass Markets*; University of California: Berkeley, CA, USA, 2017.
32. Polozov, O.; Gulwani, S. FlashMeta: A framework for inductive program synthesis. In *ACM SIGPLAN Notices*; ACM: New York, NY, USA, 2015; Volume 50, pp. 107–126.
33. Torlak, E.; Bodik, R. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Notices*; ACM: New York, NY, USA, 2014; Volume 49, pp. 530–541.
34. Wang, C.; Cheung, A.; Bodik, R. Synthesizing highly expressive SQL queries from input-output examples. In *ACM SIGPLAN Notices*; ACM: New York, NY, USA, 2017; Volume 52, pp. 452–466.
35. So, S.; Oh, H. Synthesizing imperative programs from examples guided by static analysis. In *International Static Analysis Symposium*; Springer: Cham, Switzerland, 2017; pp. 364–381.
36. Singh, R.; Gulwani, S. Synthesizing number transformations from input-output examples. In *Proceedings of the International Conference on Computer Aided Verification, July 2012*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 634–651.
37. Chen, X.; Liu, C.; Song, D. Towards synthesizing complex programs from input-output examples. *arXiv* **2017**, arXiv:1706.01284.
38. Zhang, L.; Rosenblatt, G.; Fetaya, E.; Liao, R.; Byrd, W.E.; Urtasun, R.; Zemel, R. Leveraging Constraint Logic Programming for Neural Guided Program Synthesis. In *Proceedings of the Sixth International Conference on Learning Representations, Vancouver Convention Center, Vancouver, BC, Canada, 30 April–3 May 2018*.
39. Raza, M.; Gulwani, S. Disjunctive Program Synthesis: A Robust Approach to Programming by Example. In *Proceedings of the 32nd AAAI Conference on Artificial Intelligence, New Orleans, LA, USA, 2–7 February 2018*.
40. Gulwani, S.; Harris, W.R.; Singh, R. Spreadsheet data manipulation using examples. *Commun. ACM* **2012**, *55*, 97–105. [\[CrossRef\]](#)
41. Peleg, H.; Shoham, S.; Yahav, E. Programming not only by example. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), May 2018*; IEEE: Piscataway Township, NJ, USA, 2018; pp. 1114–1124.
42. Kalyan, A.; Mohta, A.; Polozov, O.; Batra, D.; Jain, P.; Gulwani, S. Neural-guided deductive search for real-time program synthesis from examples. *arXiv* **2018**, arXiv:1804.01186.
43. Subhagit, R. From concrete examples to heap manipulating programs. In *International Static Analysis Symposium*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 126–149.
44. Le, V.; Gulwani, S. FlashExtract: A framework for data extraction by examples. In *ACM SIGPLAN Notices*; ACM: New York, NY, USA, 2014; Volume 49, pp. 542–553.
45. Lee, M.; So, S.; Oh, H. Synthesizing regular expressions from examples for introductory automata assignments. In *ACM SIGPLAN Notices*; ACM: New York, NY, USA, 2016; Volume 52, pp. 70–80.
46. Bartoli, A.; Davanzo, G.; De Lorenzo, A.; Mauri, M.; Medvet, E.; Sorio, E. Automatic generation of regular expressions from examples with genetic programming. In *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation, July 2012*; ACM: New York, NY, USA, 2012; pp. 1477–1478.
47. Gulwani, S. Automating string processing in spreadsheets using input-output examples. In *ACM Sigplan Notices*; ACM: New York, NY, USA, 2011; Volume 46, pp. 317–330.
48. Torlak, E.; Bodik, R. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, October 2013*; ACM: New York, NY, USA, 2013; pp. 135–152.
49. Albarghouthi, A.; Gulwani, S.; Kincaid, Z. Recursive program synthesis. In *International Conference on Computer Aided Verification, July 2013*; Springer: Berlin/Heidelberg, Germany, 2013; pp. 934–950.
50. Zamani, M.; Esfahani, P.M.; Majumdar, R.; Abate, A.; Lygeros, J. Symbolic control of stochastic systems via approximately bisimilar finite abstractions. *IEEE Trans. Autom. Control* **2014**, *59*, 3135–3150. [\[CrossRef\]](#)
51. Malik, S.; Weissenbacher, G. Boolean satisfiability solvers: Techniques & extensions. In *Software Safety & Security Tools for Analysis and Verification*; IOS Press: Amsterdam, The Netherlands, 2012.
52. Osera, P.M.; Zdancewic, S. Type-and-example-directed program synthesis. In *ACM SIGPLAN Notices*; ACM: New York, NY, USA, 2015; Volume 50, pp. 619–630.
53. Balog, M.; Gaunt, A.L.; Brockschmidt, M.; Nowozin, S.; Tarlow, D. Deepcoder: Learning to write programs. *arXiv* **2016**, arXiv:1611.01989.
54. Amodio, M.; Chaudhuri, S.; Repts, T. Neural attribute machines for program generation. *arXiv* **2017**, arXiv:1705.09231.



55. Udupa, A.; Raghavan, A.; Deshmukh, J.V.; Mador-Haim, S.; Martin, M.M.; Alur, R. TRANSIT: Specifying protocols with concolic snippets. In *ACM SIGPLAN Notices*; ACM: New York, NY, USA, 2013; Volume 48, pp. 287–296.
56. Liu, E.Z.; Guu, K.; Pasupat, P.; Shi, T.; Liang, P. Reinforcement learning on web interfaces using workflow-guided exploration. *arXiv* **2018**, arXiv:1802.08802.
57. Shin, R.; Polosukhin, I.; Song, D. Improving neural program synthesis with inferred execution traces. In *Proceedings of the Advances in Neural Information Processing Systems*, Palais des Congrès de Montréal, Montréal, QC, Canada, 3–8 December 2018; pp. 8917–8926.
58. Ellis, K.; Ritchie, D.; Solar-Lezama, A.; Tenenbaum, J. Learning to infer graphics programs from hand-drawn images. In *Proceedings of the Advances in Neural Information Processing Systems*, Palais des Congrès de Montréal, Montréal, QC, Canada, 3–8 December 2018; pp. 6059–6068.
59. Ganin, Y.; Kulkarni, T.; Babuschkin, I.; Eslami, S.M.; Vinyals, O. Synthesizing programs for images using reinforced adversarial learning. *arXiv* **2018**, arXiv:1804.01118.
60. Srivastava, S.; Gulwani, S.; Foster, J.S. From program verification to program synthesis. In *ACM Sigplan Notices*; ACM: New York, NY, USA, 2010; Volume 45, pp. 313–326.
61. Heule, S.; Schkufza, E.; Sharma, R.; Aiken, A. Stratified synthesis: Automatically learning the x86–64 instruction set. In *ACM SIGPLAN Notices*; ACM: New York, NY, USA, 2016; Volume 51, pp. 237–250.
62. Lau, T.; Domingos, P.; Weld, D.S. Learning programs from traces using version space algebra. In *Proceedings of the 2nd International Conference on Knowledge Capture, October 2003*; ACM: New York, NY, USA, 2003; pp. 36–43.
63. Zhang, X.; Gupta, R. Whole execution traces and their applications. *ACM Trans. Archit. Code Optim.* **2005**, *2*, 301–334. [[CrossRef](#)]
64. Bhansali, S.; Chen, W.K.; De Jong, S.; Edwards, A.; Murray, R.; Drinić, M.; Chau, J. Framework for instruction-level tracing and analysis of program executions. In *Proceedings of the 2nd International Conference on Virtual Execution Environments, June 2006*; ACM: New York, NY, USA, 2006; pp. 154–163.
65. Wang, X.; Dillig, I.; Singh, R. Program synthesis using abstraction refinement. *Proc. ACM Program. Lang.* **2017**, *63*, 1–30. [[CrossRef](#)]
66. Reyna, J. From program synthesis to optimal program synthesis. In *Proceedings of the 2011 Annual Meeting of the North American Fuzzy Information Processing Society, March 2011*; IEEE: Piscataway Township, NJ, USA, 2011; pp. 1–6.
67. Bornholt, J.; Torlak, E.; Ceze, L.; Grossman, D. Approximate Program Synthesis. In *Proceedings of the Workshop on Approximate Computing Across the Stack (WAX w/PLDI)*, Portland, OR, USA, 13 June 2015.
68. Lee, V.T.; Alaghi, A.; Ceze, L.; Oskin, M. Stochastic Synthesis for Stochastic Computing. *arXiv* **2018**, arXiv:1810.04756.
69. Lee, W.; Heo, K.; Alur, R.; Naik, M. Accelerating search-based program synthesis using learned probabilistic models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, June 2018*; ACM: New York, NY, USA, 2018; pp. 436–449.
70. Itzhaky, S.; Singh, R.; Solar-Lezama, A.; Yessenov, K.; Lu, Y.; Leiserson, C.; Chowdhury, R. Deriving divide-and-conquer dynamic programming algorithms using solver-aided transformations. In *ACM SIGPLAN Notices*; ACM: New York, NY, USA, 2016; Volume 51, pp. 145–164.
71. Pu, Y.; Bodik, R.; Srivastava, S. Synthesis of first-order dynamic programming algorithms. In *ACM SIGPLAN Notices*; ACM: New York, NY, USA, 2011; Volume 46, pp. 83–98.
72. Färm, P.; Dubrova, E.; Kuehlmann, A. Integrated logic synthesis using simulated annealing. In *Proceedings of the 21st Edition of the Great Lakes Symposium on Great Lakes Symposium on VLSI, May 2011*; ACM: New York, NY, USA, 2011; pp. 407–410.
73. Forstenlechner, S.; Fagan, D.; Nicolau, M.; O'Neill, M. Towards understanding and refining the general program synthesis benchmark suite with genetic programming. In *Proceedings of the 2018 IEEE Congress on Evolutionary Computation (CEC), July 2018*; IEEE: Piscataway Township, NJ, USA, 2018; pp. 1–6.
74. Oliveira, C.A.; Pardalos, P.M. A survey of combinatorial optimization problems in multicast routing. *Comput. Oper. Res.* **2005**, *32*, 1953–1981. [[CrossRef](#)]
75. Torlak, E.; Vaziri, M.; Dolby, J. MemSAT: Checking axiomatic specifications of memory models. *ACM Sigplan Notices* **2010**, *45*, 341–350. [[CrossRef](#)]

76. Inala, J.P.; Gao, S.; Kong, S.; Solar-Lezama, A. REAS: Combining Numerical Optimization with SAT Solving. *arXiv* **2018**, arXiv:1802.04408.
77. De Moura, L.; Bjørner, N. Z3: An efficient SMT solver. In *Proceedings of the International conference on Tools and Algorithms for the Construction and Analysis of Systems, March 2008*; Springer: Berlin/Heidelberg, Germany, 2008; pp. 337–340.
78. Reynolds, A.; Tinelli, C. SyGuS Techniques in the Core of an SMT Solver. *arXiv* **2017**, arXiv:1711.10641. [[CrossRef](#)]
79. Marić, F. Formalization and implementation of modern SAT solvers. *J. Autom. Reason.* **2009**, *43*, 81–119. [[CrossRef](#)]
80. Alexandru, C.V. Guided code synthesis using deep neural networks. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, November 2016*; ACM: New York, NY, USA, 2016; pp. 1068–1070.
81. Alexandru, C.V.; Panichella, S.; Gall, H.C. Replicating parser behavior using neural machine translation. In *Proceedings of the 25th International Conference on Program Comprehension, May 2017*; IEEE: Piscataway Township, NJ, USA, 2017; pp. 316–319.
82. Devlin, J.; Uesato, J.; Bhupatiraju, S.; Singh, R.; Mohamed, A.R.; Kohli, P. Robustfill: Neural program learning under noisy I/O. In *Proceedings of the 34th International Conference on Machine Learning, Sydney, Australia, 6–11 August 2017*; Volume 70, pp. 990–998.
83. Ke, Y.; Stolee, K.T.; Le Goues, C.; Brun, Y. Repairing programs with semantic code search (t). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), November 2015*; IEEE: Piscataway Township, NJ, USA, 2015; pp. 295–306.
84. Yang, Y.; Jiang, Y.; Gu, M.; Sun, J.; Gao, J.; Liu, H. A language model for statements of software code. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), October 2017*; IEEE: Piscataway Township, NJ, USA, 2017; pp. 682–687.
85. Alur, R.; Bodik, R.; Juniwal, G.; Martin, M.M.; Raghothaman, M.; Seshia, S.A.; Udupa, A. Syntax-guided synthesis. In *Proceedings of the 2013 Formal Methods in Computer-Aided Design, October 2013*; IEEE: Piscataway Township, NJ, USA, 2013; pp. 1–8.
86. Rolim, R.; Soares, G.; D’Antoni, L.; Polozov, O.; Gulwani, S.; Gheyi, R.; Hartmann, B. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering, May 2017*; IEEE: Piscataway Township, NJ, USA, 2017; pp. 404–415.
87. Wang, X.; Anderson, G.; Dillig, I.; McMillan, K.L. Learning Abstractions for Program Synthesis. In *Proceedings of the International Conference on Computer Aided Verification, July 2018*; Springer: Cham, Switzerland, 2018; pp. 407–426.
88. Srivastava, S.; Gulwani, S.; Foster, J.S. Template-based program verification and program synthesis. *Int. J. Softw. Tools Technol. Transf.* **2013**, *15*, 497–518. [[CrossRef](#)]
89. Abid, N.; Dragan, N.; Collard, M.L.; Maletic, J.I. The evaluation of an approach for automatic generated documentation. In *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), September 2017*; IEEE: Piscataway Township, NJ, USA, 2017; pp. 307–317.
90. Castrillon, J.; Sheng, W.; Leupers, R. Trends in embedded software synthesis. In *Proceedings of the 2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, July 2011*; IEEE: Piscataway Township, NJ, USA, 2011; pp. 347–354.
91. Kitzelmann, E.; Schmid, U. Inductive synthesis of functional programs: An explanation based generalization approach. *J. Mach. Learn. Res.* **2006**, *7*, 429–454.
92. Liang, Z.; Tsushima, K. Component-based Program Synthesis in OCaml. 2017.
93. Feng, Y.; Martins, R.; Wang, Y.; Dillig, I.; Reps, T.W. Component-based synthesis for complex APIs. *ACM SIGPLAN Notices* **2017**, *52*, 599–612. [[CrossRef](#)]
94. Jha, S.; Gulwani, S.; Seshia, S.A.; Tiwari, A. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, May 2010*; ACM: New York, NY, USA, 2010; Volume 1, pp. 215–224.
95. Jha, S.; Seshia, S.A. A theory of formal synthesis via inductive learning. *Acta Inform.* **2017**, *54*, 693–726. [[CrossRef](#)]

96. Kitzelmann, E. Inductive programming: A survey of program synthesis techniques. In *Proceedings of the International Workshop on Approaches and Applications of Inductive Programming, September 2009*; Springer: Berlin/Heidelberg, Germany, 2009; pp. 50–73.
97. Gulwani, S.; Hernández-Orallo, J.; Kitzelmann, E.; Muggleton, S.H.; Schmid, U.; Zorn, B. Inductive programming meets the real world. *Commun. ACM* **2015**, *58*, 90–99. [\[CrossRef\]](#)
98. Goffi, A.; Gorla, A.; Mattavelli, A.; Pezzè, M.; Tonella, P. Search-based synthesis of equivalent method sequences. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, November 2014*; ACM: New York, NY, USA, 2014; pp. 366–376.
99. Korukhova, Y. An approach to automatic deductive synthesis of functional programs. *Ann. Math. Artif. Intell.* **2007**, *50*, 255–271. [\[CrossRef\]](#)
100. Hofmann, M.; Kitzelmann, E.; Schmid, U. A unifying framework for analysis and evaluation of inductive programming systems. In *Proceedings of the Second Conference on Artificial General Intelligence, Atlantis, Catalonia, Spain, 16–22 July 2009*; pp. 55–60.
101. Gulwani, S. Dimensions in program synthesis. In *Proceedings of the 12th international ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming, July 2010*; ACM: New York, NY, USA, 2010; pp. 13–24.
102. Gulwani, S.; Polozov, O.; Singh, R. Program synthesis. *Found. Trends Program. Lang.* **2017**, *4*, 1–119. [\[CrossRef\]](#)
103. Allamanis, M.; Barr, E.T.; Bird, C.; Sutton, C. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, November 2014*; ACM: New York, NY, USA, 2014; pp. 281–293.
104. Li, J.; Wang, Y.; Lyu, M.R.; King, I. Code completion with neural attention and pointer networks. *arXiv* **2017**, arXiv:1711.09573.
105. Bhoopchand, A.; Rocktäschel, T.; Barr, E.; Riedel, S. Learning python code suggestion with a sparse pointer network. *arXiv* **2016**, arXiv:1611.08307.
106. Dam, H.K.; Tran, T.; Pham, T. A deep language model for software code. *arXiv* **2016**, arXiv:1608.02715.
107. Liu, C.; Wang, X.; Shin, R.; Gonzalez, J.E.; Song, D. Neural Code Completion. In *Proceedings of the International Conference on Learning Representations (ICLR) 2016, San Juan, PR, USA, 2–4 May 2016*.
108. Han, S.; Wallace, D.R.; Miller, R.C. Code completion from abbreviated input. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, November 2009*; IEEE: Piscataway Township, NJ, USA, 2009; pp. 332–343.
109. Raychev, V.; Bielik, P.; Vechev, M. Probabilistic model for code with decision trees. In *ACM SIGPLAN Notices*; ACM: New York, NY, USA, 2016; Volume 51, pp. 731–747.
110. Raychev, V.; Bielik, P.; Vechev, M.; Krause, A. Learning programs from noisy data. In *ACM SIGPLAN Notices*; ACM: New York, NY, USA, 2016; Volume 51, pp. 761–774.
111. Yamamoto, T. Code suggestion of method call statements using a source code corpus. In *Proceedings of the 2017 24th Asia-Pacific Software Engineering Conference (APSEC), December 2017*; IEEE: Piscataway Township, NJ, USA, 2017; pp. 666–671.
112. Ichinco, M. A Vision for Interactive Suggested Examples for Novice Programmers. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), October 2018*; IEEE: Piscataway Township, NJ, USA, 2018; pp. 303–304.
113. Tu, Z.; Su, Z.; Devanbu, P. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, November 2014*; ACM: New York, NY, USA, 2014; pp. 269–280.
114. White, M.; Vendome, C.; Linares-Vásquez, M.; Poshyanyk, D. Toward deep learning software repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories, May 2015*; IEEE: Piscataway Township, NJ, USA, 2015; pp. 334–345.
115. Franks, C.; Tu, Z.; Devanbu, P.; Hellendoorn, V. Cacheca: A cache language model based code suggestion tool. In *Proceedings of the 37th International Conference on Software Engineering, May 2015*; IEEE: Piscataway Township, NJ, USA, 2015; Volume 2, pp. 705–708.
116. Wan, Y.; Zhao, Z.; Yang, M.; Xu, G.; Ying, H.; Wu, J.; Yu, P.S. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, September 2018*; ACM: New York, NY, USA, 2018; pp. 397–407.

117. Hu, X.; Li, G.; Xia, X.; Lo, D.; Lu, S.; Jin, Z. Summarizing source code with transferred api knowledge. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18)*, Stockholm, Sweden, 13–19 July 2018; pp. 2269–2275.
118. Zeng, L.; Zhang, X.; Wang, T.; Li, X.; Yu, J.; Wang, H. Improving code summarization by combining deep learning and empirical knowledge (S). In *Proceedings of the SEKE*, Redwood City, CA, USA, 1–3 July 2018.
119. Rai, S.; Gaikwad, T.; Jain, S.; Gupta, A. Method Level Text Summarization for Java Code Using Nano-Patterns. In *Proceedings of the 2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, December 2017; IEEE: Piscataway Township, NJ, USA, 2017; pp. 199–208.
120. Ghofrani, J.; Mohseni, M.; Bozorgmehr, A. A conceptual framework for clone detection using machine learning. In *Proceedings of the 2017 IEEE 4th International Conference on Knowledge-Based Engineering and Innovation (KBEI)*, December 2017; IEEE: Piscataway Township, NJ, USA, 2017; pp. 0810–0817.
121. Panichella, S. Summarization techniques for code, change, testing, and user feedback. In *Proceedings of the 2018 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST)*, March 2018; IEEE: Piscataway Township, NJ, USA, 2018; pp. 1–5.
122. Malhotra, M.; Chhabra, J.K. Class Level Code Summarization Based on Dependencies and Micro Patterns. In *Proceedings of the 2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT)*, April 2018; IEEE: Piscataway Township, NJ, USA, 2018; pp. 1011–1016.
123. Hassan, M.; Hill, E. Toward automatic summarization of arbitrary java statements for novice programmers. In *Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, September 2018; IEEE: Piscataway Township, NJ, USA, 2018; pp. 539–543.
124. Decker, M.J.; Newman, C.D.; Collard, M.L.; Guarnera, D.T.; Maletic, J.I. A Timeline Summarization of Code Changes. In *Proceedings of the 2018 IEEE Third International Workshop on Dynamic Software Documentation (DySDoc3)*, September 2018; IEEE: Piscataway Township, NJ, USA, 2018; pp. 9–10.
125. Iyer, S.; Konstantas, I.; Cheung, A.; Zettlemoyer, L. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Berlin, Germany, 7–12 August 2016; pp. 2073–2083.
126. Hu, X.; Li, G.; Xia, X.; Lo, D.; Jin, Z. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, May 2018; ACM: New York, NY, USA, 2018; pp. 200–210.
127. Amodio, M.; Chaudhuri, S.; Reps, T.W. Neural Attribute Machines for Program Generation. *CoRR* abs/1705.09231. *arXiv* **2017**, arXiv:1705.09231.
128. Sulir, M.; Porubán, J. Source code documentation generation using program execution. *Information* **2017**, *8*, 148. [[CrossRef](#)]
129. Le Moulec, G.; Blouin, A.; Gouranton, V.; Arnaldi, B. Automatic production of end user documentation for DSLs. *Comput. Lang. Syst. Struct.* **2018**, *54*, 337–357. [[CrossRef](#)]
130. Louis, A.; Dash, S.K.; Barr, E.T.; Sutton, C. Deep learning to detect redundant method comments. *arXiv* **2018**, arXiv:1806.04616.
131. Wong, E.; Liu, T.; Tan, L. Clocom: Mining existing source code for automatic comment generation. In *Proceedings of the 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Montreal, QC, Canada, 2–6 March 2015; pp. 380–389.
132. Thayer, K. Using Program Analysis to Improve API Learnability. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, August 2018; ACM: New York, NY, USA, 2018; pp. 292–293.
133. Newman, C.; Dragan, N.; Collard, M.L.; Maletic, J.; Decker, M.; Guarnera, D.; Abid, N. Automatically Generating Natural Language Documentation for Methods. In *Proceedings of the 2018 IEEE Third International Workshop on Dynamic Software Documentation (DySDoc3)*, September 2018; IEEE: Piscataway Township, NJ, USA, 2018; pp. 1–2.
134. Ishida, Y.; Arimatsu, Y.; Kaixie, L.; Takagi, G.; Noda, K.; Kobayashi, T. Generating an interactive view of dynamic aspects of API usage examples. In *Proceedings of the 2018 IEEE Third International Workshop on Dynamic Software Documentation (DySDoc3)*, September 2018; IEEE: Piscataway Township, NJ, USA, 2018; pp. 13–14.
135. Yildiz, E.; Ekin, E. Creating Important Statement Type Comments in Autocomment: Automatic Comment Generation Framework. In *Proceedings of the 2018 3rd International Conference on Computer Science and Engineering (UBMK)*, September 2018; IEEE: Piscataway Township, NJ, USA, 2018; pp. 642–647.



136. Beltramelli, T. pix2code: Generating code from a graphical user interface screenshot. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems, June 2018*; ACM: New York, NY, USA, 2018; p. 3.
137. Cummins, C.; Petoumenos, P.; Wang, Z.; Leather, H. Synthesizing benchmarks for predictive modeling. In *Proceedings of the 2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), February 2017*; IEEE: Piscataway Township, NJ, USA, 2017; pp. 86–99.
138. Chen, X.; Liu, C.; Song, D. Tree-to-tree neural networks for program translation. In *Proceedings of the Advances in Neural Information Processing Systems, Palais des Congrès de Montréal, Montréal, QC, Canada, 3–8 December 2018*; pp. 2547–2557.
139. Bunel, R.; Hausknecht, M.; Devlin, J.; Singh, R.; Kohli, P. Leveraging grammar and reinforcement learning for neural program synthesis. *arXiv* **2018**, arXiv:1805.04276.
140. Barone, A.V.M.; Sennrich, R. A parallel corpus of Python functions and documentation strings for automated code documentation and code generation. *arXiv* **2017**, arXiv:1707.02275.
141. Aggarwal, K.; Salameh, M.; Hindle, A. Using Machine Translation for Converting Python 2 to Python 3 Code (No. e1817). Available online: <https://peerj.com/preprints/1459.pdf> (accessed on 3 April 2020).
142. Yin, P.; Neubig, G. A syntactic neural model for general-purpose code generation. *arXiv* **2017**, arXiv:1704.01696.
143. Puschel, M.; Moura, J.M.; Johnson, J.R.; Padua, D.; Veloso, M.M.; Singer, B.W.; Chen, K. SPIRAL: Code generation for DSP transforms. *Proc. IEEE* **2005**, *93*, 232–275. [[CrossRef](#)]
144. Strecker, M. Formal verification of a Java compiler in Isabelle. In *Proceedings of the International Conference on Automated Deduction, Copenhagen, Denmark, 27–30 July 2002*; pp. 63–77.
145. Sethi, A.; Sankaran, A.; Panwar, N.; Khare, S.; Mani, S. DLPaper2Code: Auto-generation of code from deep learning research papers. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, LA, USA, 2–7 February 2018*.
146. Subahi, A.F.; Simons, A.J. A multi-level transformation from conceptual data models to database scripts using Java agents. In *Proceedings of the 2nd Workshop of Composition of Model Transformations, Kings Collage London, London, UK, 30 September 2011*; pp. 1–7.
147. Alhefdhi, A.; Dam, H.K.; Hata, H.; Ghose, A. Generating Pseudo-Code from Source Code Using Deep Learning. In *Proceedings of the 2018 25th Australasian Software Engineering Conference (ASWEC), November 2018*; IEEE: Piscataway Township, NJ, USA, 2018; pp. 21–25.
148. Durieux, T.; Monperrus, M. Dynamoth: Dynamic code synthesis for automatic program repair. In *Proceedings of the 2016 IEEE/ACM 11th International Workshop in Automation of Software Test (AST), May 2016*; IEEE: Piscataway Township, NJ, USA, 2016; pp. 85–91.
149. Campbell, J.C.; Hindle, A.; Amaral, J.N. Syntax errors just aren't natural: Improving error reporting with language models. In *Proceedings of the 11th Working Conference on Mining Software Repositories, May 2014*; ACM: New York, NY, USA, 2014; pp. 252–261.
150. Nguyen, T.; Weimer, W.; Kapur, D.; Forrest, S. Connecting program synthesis and reachability: Automatic program repair using test-input generation. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems, April 2017*; Springer: Berlin/Heidelberg, Germany, 2017; pp. 301–318.
151. Wang, K.; Singh, R.; Su, Z. Dynamic neural program embedding for program repair. *arXiv* **2017**, arXiv:1711.07163.
152. Shin, R.; Polosukhin, I.; Song, D. Towards Specification-Directed Program Repair. In *Proceedings of the Sixth International Conference on Learning Representations (ICLR), Vancouver Convention Center, Vancouver, BC, Canada, 30 April–3 May 2018*.
153. Bhatia, S.; Singh, R. Automated correction for syntax errors in programming assignments using recurrent neural networks. *arXiv* **2016**, arXiv:1603.06129.
154. Qi, Y.; Mao, X.; Lei, Y.; Dai, Z.; Wang, C. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering, May 2014*; ACM: New York, NY, USA, 2014; pp. 254–265.
155. De Souza, E.F.; Goues, C.L.; Camilo-Junior, C.G. A novel fitness function for automated program repair based on source code checkpoints. In *Proceedings of the Genetic and Evolutionary Computation Conference, July 2018*; ACM: New York, NY, USA, 2018; pp. 1443–1450.

156. White, M.; Tufano, M.; Martinez, M.; Monperrus, M.; Poshyvanyk, D. Sorting and transforming program repair ingredients via deep learning code similarities. In *Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), February 2019*; IEEE: Piscataway Township, NJ, USA, 2019; pp. 479–490.
157. Martinez, M.; Monperrus, M. Astor: Exploring the design space of generate-and-validate program repair beyond GenProg. *J. Syst. Softw.* **2019**, *151*, 65–80. [[CrossRef](#)]
158. Hill, A.; Pasareanu, C.; Stolee, K. Poster: Automated Program Repair with Canonical Constraints. In *Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), May 2018*; IEEE: Piscataway Township, NJ, USA, 2018; pp. 339–341.
159. Martinez, M.; Monperrus, M. Astor: A program repair library for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, July 2016*; ACM: New York, NY, USA, 2016; pp. 441–444.
160. D’Antoni, L.; Samanta, R.; Singh, R. Qclose: Program repair with quantitative objectives. In *Proceedings of the International Conference on Computer Aided Verification, July 2016*; Springer: Cham, Switzerland, 2016; pp. 383–401.
161. Krawiec, K.; Bładek, I.; Swan, J.; Drake, J.H. Counterexample-driven genetic programming: Stochastic synthesis of provably correct programs. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence, Vienna, Austria, 13–19 July 2018*; pp. 5304–5308.
162. Von Essen, C.; Jobstmann, B. Program repair without regret. *Form. Methods Syst. Des.* **2015**, *47*, 26–50. [[CrossRef](#)]
163. Le, X.B.D.; Chu, D.H.; Lo, D.; Le Goues, C.; Visser, W. S3: Syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, August 2017*; ACM: New York, NY, USA, 2017; pp. 593–604.
164. Nguyen, H.D.T.; Qi, D.; Roychoudhury, A.; Chandra, S. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 35th International Conference on Software Engineering (ICSE), May 2013*; IEEE: Piscataway Township, NJ, USA, 2013; pp. 772–781.
165. Mechtaev, S.; Nguyen, M.D.; Noller, Y.; Grunske, L.; Roychoudhury, A. Semantic program repair using a reference implementation. In *Proceedings of the 40th International Conference on Software Engineering, May 2018*; ACM: New York, NY, USA, 2018; pp. 129–139.
166. Le, X.B.D.; Chu, D.H.; Lo, D.; Le Goues, C.; Visser, W. JFIX: Semantics-based repair of Java programs via symbolic PathFinder. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, July 2017*; ACM: New York, NY, USA, 2017; pp. 376–379.
167. Kneuss, E.; Koukoutos, M.; Kuncak, V. Deductive program repair. In *Proceedings of the International Conference on Computer Aided Verification, July 2015*; Springer: Cham, Switzerland, 2015; pp. 217–233.
168. Ghanbari, A.; Zhang, L. ‘PraPR: Practical Program Repair via Bytecode Mutation’. In *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), Automated Software Engineering (ASE), San Diego, CA, USA, 11–15 November 2019*; pp. 1118–1121.
169. Liu, K.; Koyuncu, A.; Kim, K.; Kim, D.; Bissyandé, T.F. LSRepair: Live search of fix ingredients for automated program repair. In *Proceedings of the 2018 25th Asia-Pacific Software Engineering Conference (APSEC), December 2018*; IEEE: Piscataway Township, NJ, USA, 2018; pp. 658–662.
170. Kaleeswaran, S.; Tulsian, V.; Kanade, A.; Orso, A. Minthint: Automated synthesis of repair hints. In *Proceedings of the 36th International Conference on Software Engineering, May 2014*; ACM: New York, NY, USA, 2014; pp. 266–276.
171. Lee, J.; Song, D.; So, S.; Oh, H. Automatic diagnosis and correction of logical errors for functional programming assignments. *Proc. ACM Program. Lang.* **2018**, *158*, 1–30. [[CrossRef](#)]
172. Roychoudhury, A. SemFix and beyond: Semantic techniques for program repair. In *Proceedings of the International Workshop on Formal Methods for Analysis of Business Systems, September 2016*; ACM: New York, NY, USA, 2016; p. 2.
173. Mechtaev, S.; Yi, J.; Roychoudhury, A. Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering, May 2015*; IEEE: Piscataway Township, NJ, USA, 2015; Volume 1, pp. 448–458.
174. Dershowitz, N.; Reddy, U.S. Deductive and inductive synthesis of equational programs. *J. Symb. Comput.* **1993**, *15*, 467–494. [[CrossRef](#)]



175. Uchihira, N.; Matsumoto, K.; Honiden, S.; Nakamura, H. Mendels: Concurrent program synthesis system using temporal logic. In *Logic Programming '87. LP, Lecture Notes in Computer Science*; Furukawa, K., Tanaka, H., Fujisaki, T., Eds.; Springer: Berlin, Heidelberg, Germany, 2005; Volume 315.
176. Manna, Z.; Waldinger, R. Fundamentals of deductive program synthesis. *IEEE Trans. Softw. Eng.* **1992**, *18*, 674–704. [[CrossRef](#)]
177. Manna, Z.; Waldinger, R. A deductive approach to program synthesis. In *Readings in Artificial Intelligence and Software Engineering*; Morgan Kaufmann: Burlington, MA, USA, 1986; pp. 3–34.
178. Shi, K.; Steinhardt, J.; Liang, P. FrAngel: Component-based synthesis with control structures. *Proc. ACM Program. Lang.* **2019**, *73*, 1–29. [[CrossRef](#)]
179. Alur, R.; Fisman, D.; Singh, R.; Solar-Lezama, A. Sygus-comp 2017: Results and analysis. *arXiv* **2017**, arXiv:1711.11438. [[CrossRef](#)]
180. Lau, T.A.; Domingos, P.M.; Weld, D.S. Version Space Algebra and its Application to Programming by Demonstration. In Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000), Stanford University, Stanford, CA, USA, 29 June–2 July 2000; pp. 527–534.
181. Subahi, A.F. Edge-Based IoT Medical Record System: Requirements, Recommendations and Conceptual Design. *IEEE Access* **2019**, *7*, 94150–94159. [[CrossRef](#)]



© 2020 by the author. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<http://creativecommons.org/licenses/by/4.0/>).